1 Python 简介

1.1 Python的诞生

Python 的作者是著名的“龟叔” Guido van Rossum (吉多·范罗苏姆)，1989年，龟叔为了打发无聊的圣诞节，开始编写 Python 语言。1991年，第一个 Python 编译器诞生。它是用 C 语言实现的，并能够调用 C 语言的库文件。

荣登2018年 TIOBE 年度最佳编程语言，很多大公司，包括 Google 、 Yahoo 都大量的使用 Python 。

1.2 提升自己的XX，你需要了解

- `Python` 的发音与拼写 `['paɪθən]`
- `Python` 的意思是蟒蛇，是取自英国20世纪70年代首播的电视喜剧《蒙提.派森干的飞行马戏团》（`Monty Python's Flying Circus`）
- `Python` 的作者是 `Guido van Rossum`（龟叔）
- `Python` 是龟叔在1989年圣诞节期间，为了打发无聊的圣诞节而用 `C` 编写的一个编程语言
- `Python` 第一个公开发行版发行于1991年
- `Python` 目前有两个主要版本，Python2和Python3
- `Life is short, you need Python`. 人生苦短，我用 `Python`。

官 网：https://www.python.org/ 中文社区：http://www.pythontab.com/

- 面向对象的解释型语言
- 简单易学
- 丰富的库
- 强制使用制表符作为语句缩进(white space)

1.3 Python主要应用领域

- 人工智能: 典型库 `NumPy`，`SciPy`，`Matplotlib`，`TensorFlow`

- 云计算: 云计算最火的语言， 典型应用 OpenStack
- WEB开发: 众多优秀的WEB框架，众多大型网站均为 Python 开发， Youtube , Dropbox , 豆瓣。。。， 典型WEB框架有 Django ， Flask
- 系统运维: 运维人员必备语言
- 金融：量化交易，金融分析，在金融工程领域， Python 不但在用，且用的最多，而且重要性逐年提高。
- 图形 GUI： PyQT , WxPython , TkInter

1.4 Python在一些公司的应用

- 谷歌： Google App Engine 、 code.google.com 、 Google earth 、谷歌爬虫、 Google 广告等项目都在大量使用 Python 开发
- CIA : 美国中情局网站就是用 Python 开发的
- NASA : 美国航天局(NASA)大量使用 Python 进行数据分析和运算
- YouTube :世界上最大的视频网站 YouTube 就是用 Python 开发的
- Dropbox :美国最大的在线云存储网站，全部用 Python 实现，每天网站处理10亿个文件的上传和下载
- Instagram :美国最大的图片分享社交网站，每天超过3千万张照片被分享，全部用 python 开发
- Facebook :大量的基础库均通过 Python 实现的
- Redhat : 世界上最流行的 Linux 发行版本中的 yum 包管理工具就是用 python 开发的
- 豆瓣: 公司几乎所有的业务均是通过 Python 开发的
- 知乎: 国内最大的问答社区，通过 Python 开发(国外 Quora)

- 除上面之外，还有搜狐、金山、腾讯、盛大、网易、百度、阿里、淘宝 、土豆、新浪、果壳等公司都在使用 Python 完成各种各样的任务。

1.5 Python的安装

1. Python (不推荐安装) 网址：https://www.python.org/
2. Anaconda (推荐安装) 网址：https://www.anaconda.com/

> - Anaconda 是一个 Python 的发行版，包括了 Python 和很多常见的软件库, 和一个包管理器 conda 。常见的科学计算类的库都包含在里面了，使得安装比常规 Python 安装要容易。
> - Anaconda 是专注于数据分析的Python发行版本，包含了 conda 、 Python 等190多个科学包及其依赖项。
> - Anaconda 的优点总结起来就八个字：**省时省心、分析利器**。
>
> **省时省心**： Anaconda 通过管理工具包、开发环境、 Python 版本，大大简化了你的工作流程。不仅可以方便地安装、更新、卸载工具包，而且安装时能自动安装相应的依赖包，同时还能使用不同的虚拟环境隔离不同要求的项目。
>
> **分析利器**： 在 Anaconda 官网中是这么宣传自己的：适用于企业级大数据分析的 Python 工具。其包含了720多个数据科学相关的开源包，在数据可视化、机器学习、深度学习等多方面都有涉及。不仅可以做数据分析，甚至可以用在大数据和人工智能领域。

安装 Anaconda 过程中需要点击 add to path ,如果忘记，需要手动配置环境变量：

```
D:\Anaconda\
D:\Anaconda\Scripts
D:\Anaconda\Library\bin
D:\Anaconda\Library\mingw-w64\bin # 可选
# 环境变量设置后，测试 conda --version 后查看是否成功
```

3. Python API 地址：https://docs.python.org/zh-cn/3/contents.html

2 Python 快速入门

2.1 Python 介绍

使用 Java 代码读取文件

```
InputStream io = null;
try {
    // 创建字节输入流对象
    io = new FileInputStream("/path/file");
    // 读取所有字节
    int b = 0;
    while ((b = io.read()) != -1) {
        System.out.print((char) b);
```

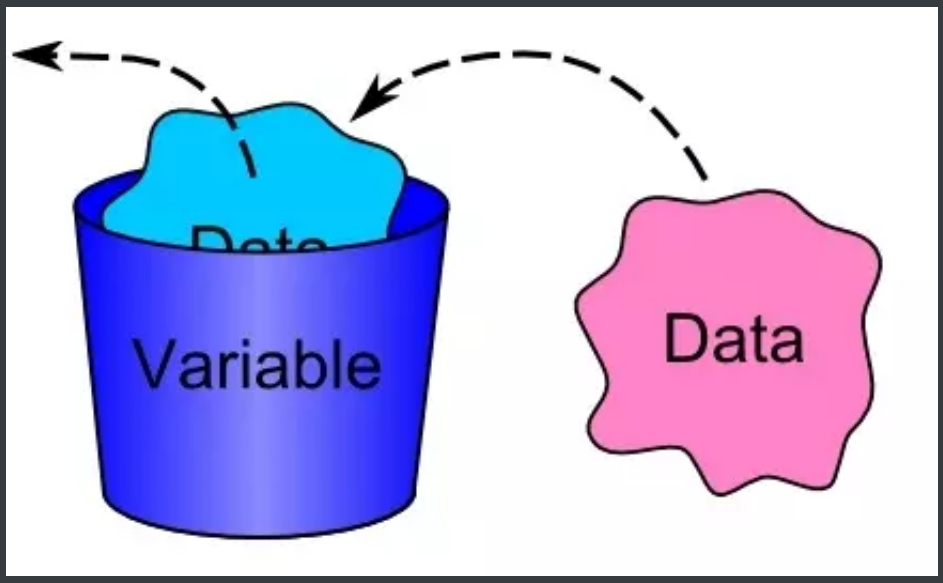
```
        }
    } catch (IOException e) {
        e.printStackTrace();
    }finally {
        try {
            // 关闭字节输入流
            io.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

使用 Python 代码读取文件

```
try:
    f = open('/path/file', 'r')
    print(f.read())
finally:
    if f:
        f.close()
# 或着使用 with， Python引入了with语句来自动帮我们调用close()方法
with open('/path/file', 'r') as f:
    print(f.read())
```

2.2 Python 数据结构

2.2.1 变量标识符



标识符就是程序员自己命名的 **变量名**。 **名字** 需要有 **见名知义** 的效果，不要随意起名。

- 标示符可以由 **字母**、**下划线** 和 **数字** 组成
- **不能以数字开头**
- **不能与关键字重名**

关键字

- **关键字** 就是在 `Python` 内部已经使用的标识符
- **关键字** 具有特殊的功能和含义
- 开发者 **不允许定义和关键字相同的名字的标示符**

通过以下命令可以查看 `Python` 中的关键字

```
import keyword
print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert',
'break', 'class', 'continue', 'def', 'del',
'elif', 'else', 'except', 'finally',
'for', 'from', 'global', 'if', 'import', 'in',
'is', 'lambda', 'nonlocal', 'not', 'or', 'pass',
'raise', 'return', 'try', 'while',
'with', 'yield']
```

> **关键字的学习及使用**，会在后面的课程中介绍
>
> - `import` **关键字** 可以导入一个 **“工具包”**
> - 在 `Python` 中不同的工具包，提供有不同的工具

1. 在定义变量时，为了保证代码格式，`=` 的左右应该各保留一个空格
2. 在 `Python` 中，如果**变量名**需要由**二个**或**多个单词**组成时，可以按照以下方式命名
    1. 每个单词都使用小写字母
    2. 单词与单词之间使用 `_` **下划线** 连接
    3. 例
       如：`first_name`、`last_name`、`qq_number`、`qq_password`

当 **变量名** 是由二个或多个单词组成时，还可以利用驼峰命名法来命名

- 小驼峰式命名法
  - 第一个单词以小写字母开始，后续单词的首字母大写
  - 例如：`firstName`、`lastName`
- 大驼峰式命名法
  - 每一个单词的首字母都采用大写字母
  - 例如：`FirstName`、`LastName`、`CamelCase`

2.2.2 变量的类型

在 `Python` 中定义变量是 **不需要指定类型**（在其他很多高级语言中都需要）

数据类型可以分为 **数字型** 和 **非数字型**

数字型

- 整型（`int`）

- 浮点型（`float`）
- 布尔型（`bool`）
  - 真 `True` `非 0 数` —— **非零即真**
  - 假 `False` `0`
- 非数字型
  - 字符串
  - 列表
  - 元组
  - 字典
- 使用 `type` 函数可以查看一个变量的类型

```
type(name)
```

不同类型变量之间的计算

1) **数字型变量** 之间可以直接计算

- 在 Python 中，两个数字型变量是可以直接进行 算数运算的
- 如果变量是 `bool` 型，在计算时
  - `True` 对应的数字是 `1`
  - `False` 对应的数字是 `0`

演练步骤

1. 定义整数 `i = 10`
2. 定义浮点数 `f = 10.5`
3. 定义布尔型 `b = True`

在PyCharm中，使用上述三个变量相互进行算术运算

```
i = 10
f = 10.5
b = True

print(i+f+b)
21.5
```

发生了自动类型转换，全部转换成了浮点数。

2) **字符串变量** 之间使用 `+` 拼接字符串

- 在 Python 中，字符串之间可以使用 `+` 拼接生成新的字符串

```
first_name = "三"
last_name = "张"
print(first_name + last_name)
```

3) **字符串变量** 可以和 **整数** 使用 `*` 重复拼接相同的字符串

```
print("-" * 50)
```

输出

```
 '--------------------------------------------------
--'
```

4) **数字型变量 和 字符串 之间 不能进行其他计算**

```
first_name = "zhang"
x = 10

print( x + first_name)
-----------------------------------------------
------------------------
TypeError: unsupported operand type(s) for +:
'int' and 'str'
类型错误：`+` 不支持的操作类型：`int` 和 `str`
```

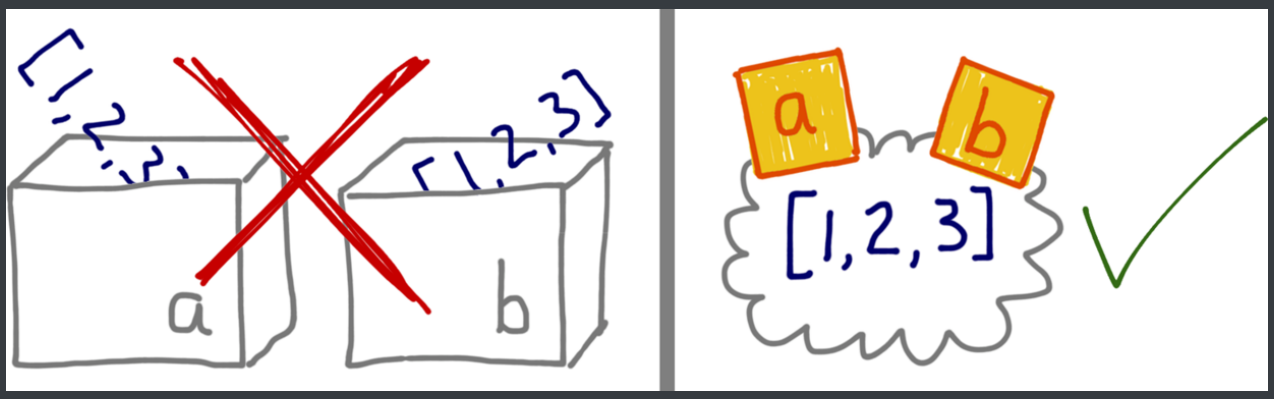
解决办法：使用str(x)将x的类型强制转换成字符串类型

```
first_name = "zhang"
x = 10
print(str(x)+first_name)
```

人们经常使用“变量是盒子”这样的比喻，但是这有碍于理解面向对象语言中的引用式变量。 Python 变量类似于 Java 中的引用式变量，因此最好把它们理解为附加在对象上的标注或便签。

在示例中所示的交互式控制台中，无法使用“变量是盒子”做解释。示意图说明了在 Python 中为什么不能使用盒子比喻，而便签则指出了变量的正确工作方式。

如果把变量想象为盒子，那么无法解释 Python 中的赋值；应该把变量视作便利贴，这样示例中的行为就好解释了



刚刚我们说明了两个变量引用同一个数据的情况，再看一种情况：一个变量先后引用不同的数据

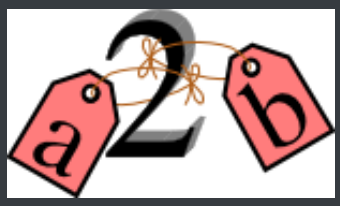
- 定义一个整数变量 `a`，并且赋值为 `1`

代码	图示
a = 1	

- 将变量 `a` 赋值为 `2`

代码	图示
a = 2	

- 定义一个整数变量 `b`，并且将变量 `a` 的值赋值给 `b`

代码	图示
b = a	

> 变量 `b` 是第 2 个贴在数字 `2` 上的标签

如果变量已经被定义，当给一个变量赋值的时候，本质上是 修改了数据的引用

- 变量 不再 对之前的数据引用
- 变量 改为 对新赋值的数据引用

2.2.3 字符串

字符串切片

获取 `Python` 字符串中的某字符可以使用索引：

```
lang = '字符串学习'
print(lang[0])
print(lang[3])
# 字
# 学
```

截取字符串中的一段字符串可以使用切片，切片在方括号中使用冒号:来分隔需要截取的首尾字符串的索引，方式是包括开头，不包括结尾

```
    lang[2:4]
    # 串学
```

当尾索引没有给出时，默认截取到字符串的末尾

```
lang[2:]
# 串学习
```

当头索引没有给出的时候默认从字符串开头开始截取

```
lang[:3]
# 字符串
```

当尾索引和头索引都没有给出的时候，默认返回整个字符串，不过这只是一个浅拷贝

```
lang[:]
# 字符串学习
```

当尾索引大于总的字符串长度时，默认只截取到字符串末尾，很明显使用这种方法来截取一段到字符串末尾的子字符串是非常不明智的，应该是不给出尾索引才是最佳实践

```
lang[3:100]
# 学习
```

当头索引为负数时，则是指从字符串的尾部开始计数，最末尾的字符记为-1，以此类推，因此此时应该注意尾索引的值，尾索引同样可以为负数，如果尾索引的值指明的字符串位置小于或等于头索引，此时返回的就是空字符串

```
    lang[-2:]
    # 学习
    lang[-2,2]
    # ''
```

切片的第三个参数是步长

```
s = 'www.python.org'

#### 切片取出来的字符串与原字符串无关，原字符串不变
print(s[6: 10])
print(s[7:: 2])
thon
# ''
```

反向取数字需要加上反向步长

```
print(s[-1: -4: -1])
print(s[-1: 2]) #### 取不到数据，因为步长默认-1
print(s[-1: 2: -1])#### 可以
gro
gro.nohtyp.
```

把字符串全部大写或小写upper， lower

```
s = 'www.PYTHON.org'
print(s.upper())    #### 全部大写
print(s.lower())    #### 全部小写
```

判读以xx开头或结尾startswith，endswith

```
s = 'www.python.org'
print(s.startswith('www'))    #### 判断是否以www开头
print(s.endswith('org'))    #### 判断是否以com结尾
True
True
```

查找元素find ， index

```
s = 'chhengt'
print(s.find('h'))  #### 通过元素找索引找到第一个就返回（可切片）
print(s.find('b'))  #### 找不到返回 -1
print(s.index('b'))   #### 找不到会报错
Traceback (most recent call last):
1
File "xxxxxxx", line 4, in <module>
-1
    print(s.index('b'))   #### 找不到会报错
ValueError: substring not found
```

strip 默认去除字符前后两端的空格， 换行符， tab

```
s = 'qqwalex qqwusir barryy'
print(s.strip('qqw'))
print(s.strip(''))
print(s.lstrip('yy'))
print(s.rstrip('yy'))
alex qqwusir barryy
qqwalex qqwusir barryy
qqwalex qqwusir barryy
qqwalex qqwusir barr
```

split 把列表分割成字符串

```
#### 分割出的元素比分隔符数+1  ***
#### 字符串变成->>>列表
s = 'qqwalex qqwusir barryy'
s1 = 'qqwale;x qqwu;sir bar;ryy'
print(s.split())    #### 默认以空格分割
print(s1.split(';'))    #### 以指定字符分割
print(s1.split(';', 1)) #### 指定分割多少个
['qqwalex', 'qqwusir', 'barryy']
['qqwale', 'x qqwu', 'sir bar', 'ryy']
['qqwale', 'x qqwu', 'sir bar;ryy']
```

join把字符串转成列表

```
#### 列表转化成字符串 list --> str
s = 'alex'####看成字符列表
li = ['aa', 'ddj', 'kk']    #### 必须全是字符串
s1 = '_'.join(s)
print(s1)
s2 = ' '.join(li)
print(s2)
a_l_e_x
aa ddj kk
```

is系列

```
#### 字符串.isalnum()    所有字符都是数字或者字母，为真返回 Ture，否则返回 False。
#### 字符串.isalpha()     所有字符都是字母，为真返回 Ture，否则返回 False。
#### 字符串.isdigit()     所有字符都是数字，为真返回 Ture，否则返回 False。
#### 字符串.islower()    所有字符都是小写，为真返回 Ture，否则返回 False。
#### 字符串.isupper()    所有字符都是大写，为真返回 Ture，否则返回 False。
#### 字符串.istitle()            所有单词都是首字母大写，为真返回 Ture，否则返回 False。
#### 字符串.isspace()    所有字符都是空白字符，为真返回 Ture，否则返回 False。

#### is 系列
name = 'python123'
print(name.isalnum())   #### 字符串由数字或字母组成时返回真
print(name.isalpha())   #### 字符只由字母组成时返回真
print(name.isdigit())   #### 字符只由数字组成时返回真
```

count 计算字符串中某个字符出现的次数

```
#### count 计算字符串中某个字符出现的次数  ***
s = 'www.python.org'
print(s.count('o'))
print(s.count('0', 7))
2
1
```

replace* 替换字符串中指定的字符

```
s = 'asdf 之一，asdf也，asdf'
#### replace  ***
s1 = s.replace('asdf', '日天')
print(s1)
s1 = s.replace('asdf', '日天', 1)
print(s1)
日天 之一，日天也，日天
日天 之一，asdf也，asdf
```

format格式化输出

```
#### format 格式化输出   ***
#### 第一种方式:
s = '我叫{}，今年{}，  性别{}'.format('小五', 25,
'女')
print(s)
#### 第二种方式
s1 = '我叫{0}，今年{1}，  性别{2}，我依然叫
{0}'.format('小五', 25, '女')
print(s1)
#### 第三种方式
s2 = '我叫{name}，今年{age}，  性别{sex}，我依然叫
{name}'.format(age=25, sex='女',name='小五')
print(s2)
我叫小五，今年25，  性别女
我叫小五，今年25，  性别女，我依然叫小五
我叫小五，今年25，  性别女，我依然叫小五
```

capitalize() 首字母大写

```
s = 'python 4fhsa￥fh。f'
#### capitalize() 首字母大写
s1 = s.capitalize()
print(s1)
Python wuang4fhsa￥fh。f
```

center() 将字符串居中可以设置总长度，可以设置填充物

```
#### center() 将字符串居中可以设置总长度，可以设置填充物
*
s = 'python'
s2 = s.center(50)
s2 = s.center(50, '*')
print(s2)
**********************python**********************
```

title 非字母隔开的每个单词的首字母大写

```
s = 'python study4fhsa￥fh。f'
#### title  非字母隔开的每个单词的首字母大写 *
s4 = s.title()
print(s4)
Python Study4Fhsa￥Fh。F
```

字符串是不可变变量，不支持直接通过下标修改

```
msg = 'abcdefg'
# msg[2] = 'z'

msg = msg[:2] + 'z' + msg[3:]

print(msg)
```

2.2.4 列表 list

使用可视化工具查看变化过程

列表 list： 列表是最常用的 Python 数据类型，它可以作为一个方括号内的逗号分隔值出现

```
#创建空列表
list1 = []
#创建带有元素的列表
list2 = [10, 20, 30, 10]
print(list2)  #[10, 20, 30, 10]
list3 = [33, "good", True, 10.32]
print(list3)  #[33, 'good', True, 10.32]
# 列表组合  语法：  list3 = list1 + list2
# 列表重复  语法：  list2 = list1 * n，如下代码：
list1 = [1, 2, 3]
print(list1*3)    #[1, 2, 3, 1, 2, 3, 1, 2, 3]
# 判断元素是否在列表中 语法：x in list1 若存在则返回True，否则返回False，如下代码：
list1 = [1, 2, 3]
print(1 in list1) #True

# 切片（Slice）操作符 语法：list1[start:end]
# 表示获取从开始下标到结束下标的所有元素[start，end)。若不指定start，则默认从0开始截取，截取到指定位置；若不指定end，则从指定位置开始截取，截取到末尾结束。如下代码：
list1 = [1, 2, 3, 'hello', 'yes', 'no']
print(list1[2:4])   #[3, 'hello']
list2 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list2[-10:0]) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] 后10个数
```

```
print(list2[:10:2]) # [0, 2, 4, 6, 8] 前10个数，每两个取一个

# list.append(元素/列表) 在列表中末尾添加新的元素【在原本的列表中追加元素】 append()中的值可以是列表也可以是普通元素
list1 = [3, 4, 6]
list1.append(6)
print(list1)    #[3, 4, 6, 6]

# list.extend(列表) 功能：在列表的末尾一次性追加另外一个列表中的多个值 注意：extend()中的值只能是列表/元组[一个可迭代对象]，不能是元素
list1 = [1,2,3]
list2 = [3, 4,5]
list1.extend(list2)
print(list1)    #[1, 2, 3, 3, 4, 5]

# list.insert(下标值，元素/列表) 功能：在下标处插入元素，不覆盖原本的数据，原数据向后顺延 注意：插入的数据可以是元素也可以为列表
list1 = [1,2,3]
list1.insert(1,0)
print(list1)    #[1, 0, 2, 3]
list1.insert(1,[2, 4, 8])
print(list1)    #[1, [2, 4, 8], 0, 2, 3]

# list.pop(下标值) 功能：移除列表中指定下标处的元素(默认移除最后一个元素)，并返回移除的数据
```

```
list1 = [1, [2, 4, 8], 0, 2, 3]
print(list1.pop())    #3
print(list1.pop(2))   #0
print(list1)      #[1, [2, 4, 8], 2]

# list.remove(元素) 功能：移除列表中的某个元素第一个匹配结果
list1 = [1, 2, 3]
list1.remove(2)
print(list1)    #[1, 3]

# list.clear() 功能：清除列表中所有的数据
list1 = [1, 2, 3]
list1.clear()
print(list1)    #[]

# list.index(object[, start][, stop])  功能：从指定的范围的列表中找出某个值第一匹配的索引值 ；若不指定范围，则默认是整个列表
list1 = [1, 2, 3]
print(list1.index(2)) #1

# list.count(元素)  功能：查看元素在列表中出现的次数
list1 = [1, 2, 3, 1]
print(list1.count(1)) #2

# len(list)  功能：  获取元素列表个数
list1 = [1, 2, 3, 1]
print(len(list1)) #4
```

```
# max(list)  功能：  获取列表中的最大值
list1 = [1, 2, 3, 1]
print(max(list1)) #3

# min(list)  功能：获取列表中的最小值
list1 = [1, 2, 3, 1]
print(min(list1)) #1

# list.reverse()  功能：将列表中的元素倒叙，操作原列表，不返回新的列表
list1 = [1, 2, 3, 1]
list1.reverse()
print(list1)    #[1, 3, 2, 1]

# list.sort(reverse=False)  功能：将list中的元素进行升序排列【默认reverse=False】，当reverse为True的时候，降序排列。
list1 = [1, 2, 3, 1]
list1.sort()
print(list1)    #[1, 1, 2, 3]

# list(元组) 功能：将元组转为列表
list1 = list((1, 2, 3, 4))
print(list1)    #[1, 2, 3, 4]
```

2.2.5 元祖 tuple

元祖 tuple： tuple 和 list 非常类似，但是 tuple 一旦初始化就不能修改。优点：查询速度快，可以二分查找，key 是不可以重复的。

```
classmates = ('Michael', 'Bob', 'Tracy')
# classmates这个tuple不能变了，它也没有append()，insert()这样的方法。其他获取元素的方法和list是一样的，你可以正常地使用classmates[0]，classmates[-1]，但不能赋值成另外的元素。
# 不可变的tuple有什么意义？因为tuple不可变，所以代码更安全。如果可能，能用tuple代替list就尽量用tuple
# Python在显示只有1个元素的tuple时，也会加一个逗号,，以免你误解成数学计算意义上的括号。
# 最后来看一个“可变的”tuple：
t = ('a', 'b', ['A', 'B'])
t[2][0] = 'X'
t[2][1] = 'Y'
print(t)  # ('a', 'b', ['X', 'Y'])
# 这个tuple定义的时候有3个元素，分别是'a'，'b'和一个list。不是说tuple一旦定义后就不可变了吗？怎么后来又变了？
# 所以，tuple所谓的“不变”是说，tuple的每个元素，指向永远不变。即指向'a'，就不能改成指向'b'，指向一个list，就不能改成指向其他对象，但指向的这个list本身是可变的！
```

2.2.6 字典 dict

字典 dict： dict全称 dictionary，在其他语言中也称为 map，使用键-值（key-value）存储，具有极快的查找速度

```
# 定义dict

dict1 = {'name': 'Bob', 'age': 18, 'gender': '男'}

# dict的删除方法

dict1.pop('age') # 18 使用pop()删除，如果有键，则删除，如果没有则会报错，如果不希望出现报错信息，可以在删除的后面添加信息
dict1.popitem()  #使用popitem()删除，随机删除，返回的是一个元组，元组里面存储的删除的键值，推荐使用pop()方法进行删除
del dict1['name'] #删除字典的某个键
del dict1 # 使用del()删除，del()可以删除整个字典

# dict的修改

dict1['name']='Jordan' # 直接修改
dict2={'address':'北京海淀','age':22}
dict1.update(dict2) # 调用update()修改

# dict 遍历

for i in dict1.keys(): # 打印dict1的键
    print(i ) # name age gender
for v in dict1.values(): # 打印dict1的值
  print(v) #Bob 18 男
for i in dict1.items(): #打印字典的键值:
```

```
  print(i) # ('name', 'Bob') ('age', 18)
('gender', '男')
for k,v in dict1.items():
  print(k,v) # name Bob age 18  gender 男

dict查询 get()方法

   dict1.get('name') #使用get()方法可以查询某个键是否存在，如果不存在此键，则会返回None，但是可以在get()方法中添加信息避免出现None
```

2.3.7 集合 set

`set`： `set`和`dict`类似，也是一组`key`的集合，但不存储`value`。由于`key`不能重复，所以，在`set`中，没有重复的`key`，要创建一个`set`，需要提供一个`list`作为输入集合

```
# 创建 set
s = set([1, 2, 3])
print(s) # ，传入的参数[1, 2, 3]是一个list，而显示的set([1, 2, 3])只是告诉你这个set内部有1，2，3这3个元素，显示的[]不表示这是一个list

# 通过add(key)方法可以添加元素到set中，可以重复添加，但不会有效果
s.add(4)

# 通过remove(key)方法可以删除元素：
s.remove(4)
```

2.3 流程控制

2.3.1 条件判断与循环

Pthon 条件语句跟其他语言基本一致的，都是通过一条或多条语句的执行结果（True 或者 False）来决定执行的代码块。

Python 程序语言指定任何非 0 和非空（None）值为 True，0 或者 None 为 False。

if 语句的基本形式

Python 中，if 语句的基本形式如下：

```
if 判断条件:
    执行语句……
else:
    执行语句……
```

if 语句多个判断条件的形式

有些时候，我们的判断语句不可能只有两个，有些时候需要多个，比如上面的例子中大于 60 的为及格，那我们还要判断大于 90 的为优秀，在 80 到 90 之间的良好呢?

这时候需要用到 if 语句多个判断条件，

用伪代码来表示：

```
if 判断条件1:
    执行语句1……
elif 判断条件2:
    执行语句2……
elif 判断条件3:
    执行语句3……
else:
    执行语句4……
```

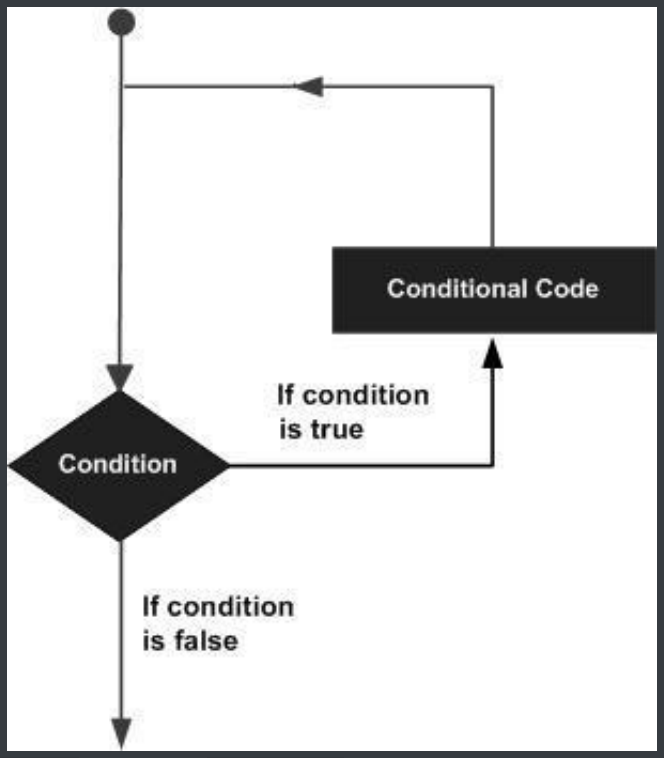
if 语句多个条件同时判断

Python 不像 Java 有 switch 语句，所以多个条件判断，只能用 elif 来实现，但是有时候需要多个条件需同时判断时，可以使用 or （或），表示两个条件有一个成立时判断条件成功；使用 and （与）时，表示只有两个条件同时成立的情况下，判断条件才成功。

循环语句

一般编程语言都有循环语句，循环语句允许我们执行一个语句或语句组多次。

循环语句的一般形式如下：



Python 提供了 for 循环和 while 循环，当然还有一些控制循环的语句：

循环控制语句	描述
break	在语句块执行过程中终止循环，并且跳出整个循环
continue	在语句块执行过程中终止当前循环，跳出该次循环，执行下一次循环
pass	pass 是空语句，是为了保持程序结构的完整性

While 循环语句

```
count = 1
sum = 0
while count <= 100:
    sum = sum + count
    count = count + 1
print(sum)
```

输出的结果：

```
5050
```

当然 `while` 语句时还有另外两个重要的命令 `continue`， `break` 来跳过循环， `continue` 用于跳过该次循环， `break` 则是用于跳出本层循环

比如，上面的例子是计算 1 到 100 所有整数的和，当我们需要判断 sum 大于 1000 的时候，不在相加时，可以用到 `break`，退出整个循环

```
count = 1
sum = 0
while  count <= 100:
    sum = sum + count
    if  sum > 1000:  #当 sum 大于 1000 的时候退出循环
        break
    count = count + 1
print(sum)
```

输出的结果：

```
1035
```

有时候，我们只想统计 1 到 100 之间的奇数和，那么也就是说当 `count` 是偶数，也就是双数的时候，我们需要跳出当次的循环，不想加，这时候可以用到 `continue`

```
count = 1
sum = 0
while  count <= 100:
    if count % 2 == 0:  # 双数时跳过输出
        count = count + 1
        continue
    sum = sum + count
    count = count + 1
print(sum)
```

输出的语句：

```
2500
```

在 Python 的 `while` 循环中，还可以使用 `else` 语句， `while … else` 在循环条件为 `false` 时执行 `else` 语句块

比如：

```
count = 0
while count < 5:
    print (count)
    count = count + 1
else:
    print (count)
```

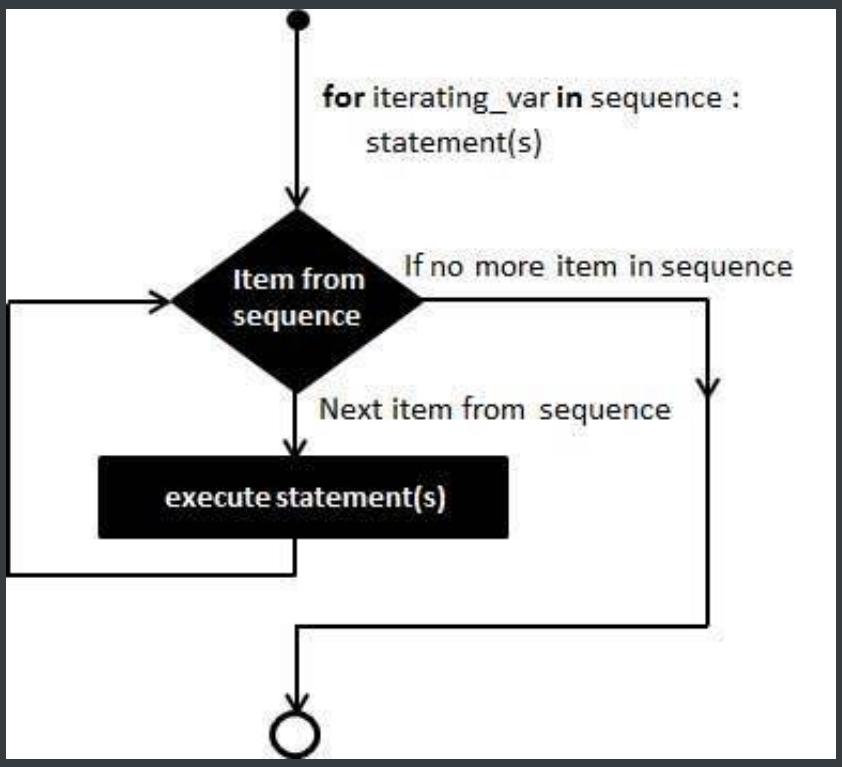
输出的结果：

```
0
1
2
3
4
5
```

for 循环语句

`for` 循环可以遍历任何序列的项目，如一个字符串

它的流程图基本如下：



基本的语法格式：

```
for iterating_var in sequence:
   statements(s)
```

实例：

```
for letter in 'www.python.org':
    print(letter)
```

输出的结果如下：

```
w
w
w
.
p
```

```
y
t
h
o
n
.
o
r
g
```

嵌套循环

Python 语言允许在一个循环体里面嵌入另一个循环。上面的实例也是使用了嵌套循环的。

具体的语法如下：

for 循环嵌套语法

```
for iterating_var in sequence:
   for iterating_var in sequence:
      statements(s)
   statements(s)
```

while 循环嵌套语法

```
while expression:
    while expression:
        statement(s)
    statement(s)
```

除此之外，你也可以在循环体内嵌入其他的循环体，如在 while 循环中可以嵌入 for 循环， 反之，你可以在 for 循环中嵌入 while 循环

有 while … else 语句，当然也有 for … else 语句啦，for 中的语句和普通的没有区别，else 中的语句会在循环正常执行完（即 for 不是通过 break 跳出而中断的）的情况下执行，while … else 也是一样。

2.3.2 Python 推导式

推导式 comprehensions （又称解析式），是 Python 的一种独有特性。推导式是可以从一个数据序列构建另一个新的数据序列。

range() 函数

Python 的 range() 函数可用来创建一个整数列表，一般用在 for 循环中

range() 语法: range(start, stop[, step])

start : 计数从 start 开始,默认是从0开始(闭区间),如: range(5) 等价于 range(0,5)

`stop`：计数到 `stop` 结束,但不包括 `stop` (开区间).如: `range(0,5)` 是[0, 1, 2, 3, 4],不包含5

`step`:步长,相邻两个值的差值,默认为1.如: `range(0,5)` 相当于 `range(0, 5, 1)`

例如 要打印数字1~5，需要使用 `range(1,6)`：

```
for value in range(1,5):
    print(value)
```

上述代码好像应该打印数字1~5，但实际上它不会打印数字5:

```
1
2
3
4
```

为什么要在推导式前讲 `range()` ,因为推导式是通过一个可迭代对象来生成的, `range()` 可以说是推导式中最常用的可迭代对象了.对推导式来说, `range()` 是其中的精髓之一.没有 `range()` 推导式的可读性和简洁性将会大打折扣

列表推导式

`for` 循环有非常广的应用场景,也可以用来创建一个列表,而列表推导式就相当于 `for` 循环创建列表的简化版

```python
# for循环
list_a = list()
for a in range(5):
  list_a.append(a)
print(list_a)

# 列表推导式
list_b = [b for b in range(5)]
print(list_b)

# in后面跟其他可迭代对象,如字符串
list_c = [7 * c for c in "python"]
print(list_c)

# 带if条件语句的列表推导式
list_d = [d for d in range(6) if d % 2 != 0]
print(list_d)

# 多个for循环
list_e = [(e, f * f) for e in range(3) for f in
range(5, 15, 5)]
print(list_e)

# 嵌套列表推导式,多个并列条件
list_g = [[x for x in range(g - 3, g)] for g in
range(22) if g % 3 == 0 and g != 0]
print(list_g)
```

2.4 Python 函数

2.4.1 函数的定义和调用

使用函数的好处：

- 代码可重用
- 保持一致性
- 可扩展性

函数是逻辑结构化和过程化的编程方法。

```
def 函数名(形参列表):
    //由零条到多条可执行语句组成的代码块
    [return [返回值]]
```

其中，用 [] 括起来的为可选择部分，即可以使用，也可以省略。

此格式中，各部分参数的含义如下：

- 函数名：从语法角度来看，函数名只要是一个合法的标识符即可；从程序的可读性角度来看，函数名应该由一个或多个有意义的单词连缀而成，每个单词的字母全部小写，单词与单词之间使用下画线分隔
- 形参列表：用于定义该函数可以接收的参数。形参列表由多个形参名组成，多个形参名之间以英文逗号（,）隔开。一旦在定义函数时指定了形参列表，调用该函数时就必须传入相应的参数值，也就是说，谁调用函数谁负责为形参赋值

```
def print_welcome(name):
    print("Welcome", name)


print_welcome("Python") # 调用函数
```

2.4.2 函数的参数

1. 形参：形参变量只有在被调用时才被分配内存单元，在调用结束时，立即释放所分配的内存单元。因此，形参只在函数内部有效，函数调用结束返回主调函数后不能继续使用该形参
2. 实参：实参可以是常量，变量，表达式，函数等，在进行函数调用时，它们必须要有却定的值（要有定义），以便把值传给形参

位置参数

位置参数，字面意思也就是按照参数的位置来进行传参，有几个位置参数在调用的时候就要传几个，否则就会报错了。

默认参数

默认参数就是在定义形参的时候，给函数默认赋一个值，比如说数据库的端口这样的，默认给它一个值，这样就算你在调用的时候没传入这个参数，它也是有值的

可变参数

1. 可变参数用*来接收，不是必传的；

2. 它把传入的元素全部都放到了一个元祖里；
3. 不显示参数个数，后面想传多少个参数就传多少个，它用在参数比较多的情况下
4. 如果位置参数、默认值参数、可变参数一起使用的的话，可变参数必须在位置参数和默认值参数后面

关键字参数

1. 关键字参数使用**来接收
2. 返回的是字典
3. 不限制参数个数，非必传
4. 当然也可以和上面的几种一起来使用，如果要一起使用的话，关键字参数必须在最后面

注意：如果一定要一起用的话，要按下顺序写，不能换顺序，否则会出错：1、位置参数；2、默认值参数；3、可变参数；4、关键字参数。

2.4.3 局部变量和全局变量

1. 局部变量意思就是在局部生效的，出了这个变量的作用域，这个变量就失效了。
2. 全局变量的意思就是在整个程序里面都生效的，在程序最前面定义的都是全局变量。
3. 全局变量如果要在函数中修改的话，需要加global关键字声明，如果是list、字典和集合的话，则不需要加global关键字，直接就可以修改。
4. 任何函数都可以修改，所以尽量少用全局变量，原因一不够安全。原因二全局变量一直占用内存。

2.4.4 递归调用

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

递归调用的意思就是，在这个函数内部自己调用自己，就有点循环的意思。

2.4.5 匿名函数

`Python` 使用 `lambda` 来创建匿名函数。

所谓匿名，意即不再使用 `def` 语句这样标准的形式定义一个函数。

- `lambda` 只是一个表达式，函数体比 `def` 简单很多。
- `lambda` 的主体是一个表达式，而不是一个代码块。仅仅能在 `lambda` 表达式中封装有限的逻辑进去
- `lambda` 函数拥有自己的命名空间，且不能访问自己参数列表之外或全局命名空间里的参数。
- `lambda` 函数看起来只能写一行，目的是调用小函数时不占用栈内存从而增加运行效率

语法:

```
lambda [arg1 [,arg2,.....argn]]:expression
```

```
# 可写函数说明
sum = lambda arg1, arg2: arg1 + arg2

# 调用sum函数
print ("相加后的值为 : ", sum( 10, 20 ))
print ("相加后的值为 : ", sum( 20, 20 ))
```

3 模块

如果你从 Python 解释器退出并再次进入，之前的定义（函数和变量）都会丢失。因此，如果你想编写一个稍长些的程序，最好使用文本编辑器为解释器准备输入并将该文件作为输入运行。这被称作编写 脚本 。随着程序变得越来越长，你或许会想把它拆分成几个文件，以方便维护。你亦或想在不同的程序中使用一个便捷的函数， 而不必把这个函数复制到每一个程序中去。

为支持这些， Python 有一种方法可以把定义放在一个文件里，并在脚本或解释器的交互式实例中使用它们。这样的文件被称作 模块 ；模块中的定义可以 导入 到其它模块或者 主 模块（你在顶级和计算器模式下执行的脚本中可以访问的变量集合）。

模块是一个包含 Python 定义和语句的文件。文件名就是模块名后跟文件后缀 `.py` 。在一个模块内部，模块名（作为一个字符串）可以通过全局变量 `__name__` 的值获得。

3.1 import 语句

想使用 Python 源文件，只需在另一个源文件里执行 import 语句，语法如下：

```
import module1[, module2[,... moduleN]
```

当解释器遇到 import 语句，如果模块在当前的搜索路径就会被导入。

搜索路径是一个解释器会先进行搜索的所有目录的列表。如想要导入模块 support，需要把命令放在脚本的顶端：

```
# Filename: support.py

def print_func( name ):
    print ("Hello : ", name)
    return
```

test.py 引入 support 模块：

```
# 导入模块
import support

# 现在可以调用模块里包含的函数了
support.print_func("张三")
```

一个模块只会被导入一次，不管你执行了多少次 import。这样可以防止导入模块被一遍又一遍地执行。

当我们使用 import 语句的时候， Python 解释器是怎样找到对应的文件的呢?

这就涉及到 Python 的搜索路径，搜索路径是由一系列目录名组成的， Python 解释器就依次从这些目录中去寻找所引入的模块。

这看起来很像环境变量，事实上，也可以通过定义环境变量的方式来确定搜索路径。

3.2 from … import 语句

Python 的 from 语句让你从模块中导入一个指定的部分到当前命名空间中，语法如下:

```
from modname import name1[, name2[, ... nameN]]
```

创建一个 fibo.py 的文件

```
# 斐波那契(fibonacci)数列模块

def fib(n):    # 定义到 n 的斐波那契数列
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # 返回到 n 的斐波那契数列
    result = []
```

```
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

例如，要导入模块 fibo 的 fib 函数，使用如下语句：

```
from fibo import fib, fib2
fib(500)
# 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这个声明不会把整个 fibo 模块导入到当前的命名空间中，它只会将 fibo 里的 fib 函数引入进来。

3.3 __name__ 属性

一个模块被另一个程序第一次引入时，其主程序将运行。如果我们想在模块被引入时，模块中的某一程序块不执行，我们可以用 __name__ 属性来使该程序块仅在该模块自身运行时执行。

```
# Filename: using_name.py

if __name__ == '__main__':
    print('程序自身在运行')  # 程序自身在运行
else:
    print('我来自另一模块')
```

```
import using_name
# 我来自另一模块
```

3.4 作用域

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在 Python 中，是通过 _ 前缀来实现的。

正常的函数和变量名是公开的（public），可以被直接引用，比如：`abc`，`x123`，`PI`等；

类似 `__xxx__` 这样的变量是特殊变量，可以被直接引用，但是有特殊用途，比如上面的 `__author__`，`__name__` 就是特殊变量，`hello` 模块定义的文档注释也可以用特殊变量 `__doc__` 访问，我们自己的变量一般不要用这种变量名；

类似 `_xxx` 和 `__xxx` 这样的函数或变量就是非公开的（private），不应该被直接引用，比如 `_abc`，`__abc` 等；

之所以我们说，private函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为Python并没有一种方法可以完全限制访问private函数或变量，但是，从编程习惯上不应该引用private函数或变量。

```
def _private_1(name):
    return 'Hello, %s' % name

def _private_2(name):
    return 'Hi, %s' % name

def greeting(name):
    if len(name) > 3:
        return _private_1(name)
    else:
        return _private_2(name)
```

我们在模块里公开 greeting() 函数，而把内部逻辑用private函数隐藏起来了，这样，调用 greeting() 函数不用关心内部的private函数细节，这也是一种非常有用的代码封装和抽象的方法，即：

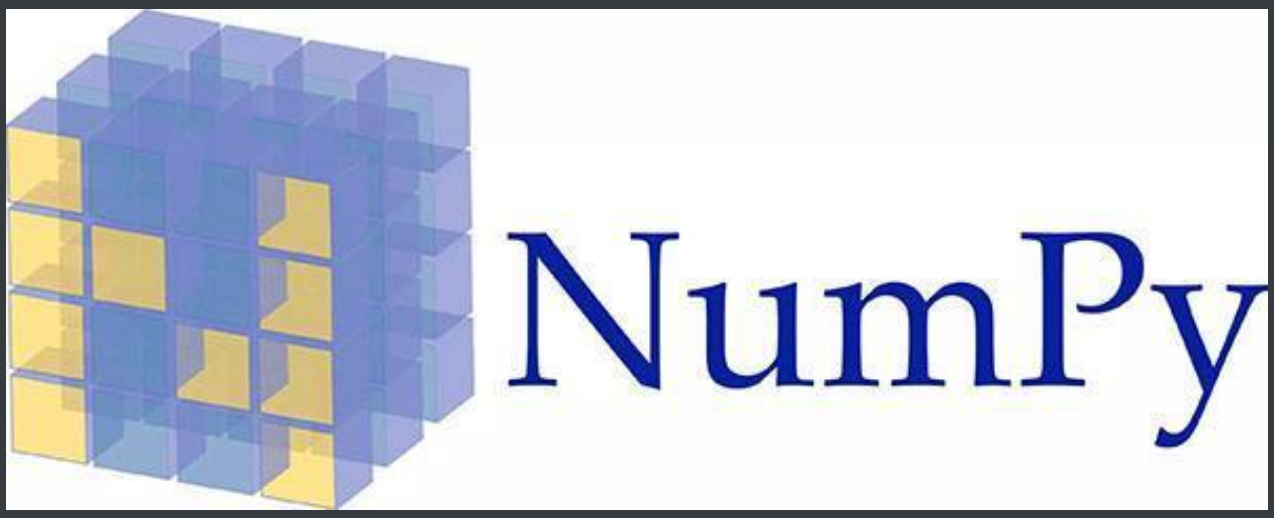
外部不需要引用的函数全部定义成 private ，只有外部需要引用的函数才定义为 public 。

4 Python 数据分析

4.1 Numpy

4.1.1 Numpy概述

NumPy （Numerical Python 的简称）是 Python 数值计算最重要的基础包。大多数提供科学计算的包都是用 NumPy 的数组作为构建基础。



Why NumPy?

- 一个强大的N维数组对象 ndarray， 具有矢量算术运算和复杂广播能力的快速且节省空间的多维数组
- 用于集成由 C 、 C++ 、 Fortran 等语言类库的 C 语言 API
- 线性代数、随机数生成以及傅里叶变换功能。
- 用于对整组数据进行快速运算的标准数学函数（无需编写循环）,支持大量的数据运算
- 是众多机器学习框架的基础库

> Tips： Python 的面向数组计算可以追溯到1995年， Jim Hugunin 创建了 Numeric 库。接下来的10年，许多科学编程社区纷纷开始使用 Python 的数组编程，但是进入21世纪，库的生态系统变得碎片化了。2005年， Travis Oliphant 从 Numeric 和 Numarray 项目整了出了 NumPy 项目，进而所有社区都集合到了

> 这个框架下。

NumPy之于数值计算特别重要的原因之一，是因为它可以高效处理大数组的数据。这是因为：

- NumPy是在一个连续的内存块中存储数据，独立于其他Python内置对象。NumPy的C语言编写的算法库可以操作内存，而不必进行类型检查或其它前期工作。比起Python的内置序列，NumPy数组使用的内存更少。
- NumPy可以在整个数组上执行复杂的计算，而不需要Python的for循环。

我们来直观感受一下numpy的运行速度：

```
# 导入numpy
# 我们将依照标准的Numpy约定，即总是使用import numpy as np
# 当然你也可以为了不写np，而直接在代码中使用from numpy import *，
# 但是建议你最好还是不要养成这样的坏习惯。
import numpy as np
# 生成一个numpy对象，  一个包含一百万整数的数组
np_arr = np.arange(1000000)
# 一个等价的Python列表：
py_list = list(range(1000000))
```

对各个序列分别平方操作

```
%time for _ in range(10): np_arr2 = np_arr ** 2
```

CPU times: user 13.1 ms, sys: 5.62 ms, total: 18.8 ms Wall time: 17.7 ms

```
%time for _ in range(10): py_list2 = [x ** 2 for x in py_list]
```

CPU times: user 3.06 s, sys: 173 ms, total: 3.24 s Wall time: 3.25 s 由上述代码可以看出，基于 NumPy 的算法要比纯 Python 快10到100倍（甚至更快），并且使用的内存更少

4.1.2 创建ndarray

NumPy 最重要的一个特点就是 ndarray（N-dimensional array），即 N 维数组），该对象是一个快速而灵活的大数据集容器。你可以利用这种数组对整块数据执行一些数学运算，其语法跟标量元素之间的运算一样。

```
# 使用array创建数组:
arr = np.array([1,2,3]) # 创建一维数组
arr = np.array([1,2,3],[4,5,6],[7,8,9])  # 创建二维数组
# 使用arange创建数组
arr = np.arange(0,10,1)  # 创建初值为0，终值为9的递增一维数组
arr = np.arange(12).reshape(3,4) # 创建1-12的3行4列的二维数组
```

```
# 使用linspace创建等差数组
arr = np.linspace(0,10,1)
# 使用logspace创建等比数组
arr = np.logspace(0,2,20)
# 使用zeros创建元素全为0的数组
arr = np.zeros((2,3)) # 创建2行3列的二维数组
# 使用eye创建对角线元素为1的单位矩阵
arr = np.eye(3)
# 使用diag创建对角线元素为0或其他的矩阵
arr = np.diag([1,2,3,4])
# 使用ones创建元素全为1的数组
arr = np.ones((4,3))
```

4.1.3 Numpy函数

数组的属性

```
arr.ndim  --数组的维度
arr.shape --数组的形状，为n行m列
arr.size  --数组的元素总数
arr.dtype --数组的元素类型
arr.itemsize --数组每个元素的大小
```

数组的数据类型

```
bool --布尔值，True或False
int8/in16/int32/int64  --整数
float16/float32/float64 --浮点数
# 数组的类型转换
np.bool(42)
np.int16(42.3)
np.float(42)
```

生成随机数：

```
np.random.random(100)  --生成100个0-1的随机数
np.random.rand(10,5)   --生成10行5列0-1的服从均匀分布的随机数
np.random.randn(10,5)  --生成正态分布的随机数
np.random.randint(2,10,size = [2,5])  --生成2行5列取值为2-10整数的随机数
```

数组的索引：

```
# 一维数组的索引，与Python中list的索引方式一致
arr[5], arr[3:5], arr[:5], arr[-1], arr[1:-1:2]--2表示隔一个元素取一个

# 二维数组的索引：arr[1:,2:],各个维度的索引用逗号隔开
```

4.2 Pandas

4.2.1 Pandas概述

Pandas （Python Data Analysis Library ）是基于 NumPy 的一种工具，该工具是为了解决数据分析任务而创建的。 Pandas 纳入了大量库和一些标准的数据模型，提供了高效地操作大型数据集所需的工具。Pandas提供了大量能使我们快速便捷地处理数据的函数和方法。你很快就会发现，它是使Python成为强大而高效的数据分析环境的重要因素之一。

Pandas 是 Python 的一个数据分析包，最初由 AQR Capital Management 于2008年4月开发，并于2009年底开源出来，目前由专注于 Python 数据包开发的 PyData 开发 team 继续开发和维护，属于 PyData 项目的一部分。 Pandas 最初被作为金融数据分析工具而开发出来。

Pandas 含有使数据分析工作变得更快更简单的高级数据结构和操作工具。 pandas 是基于 Numpy 构建的，让以 Numpy 为中心的应用变得更简单。

Pandas 专用于数据预处理和数据分析的 Python 第三方库，最适合处理大型结构化表格数据

- Pandas 是2008年 Wes McKinney 于 AQR 资本做量化分析师时创建
- Pandas 借鉴了 R 的数据结构
- Pandas 基于 Numpy 搭建，支持 Numpy 中定义的大部分计算
- Pandas 含有使数据分析工作更简单高效的高级数据结构和操作工具
- Pandas 底层用 Cython 和 C 做了速度优化，极大提高了执行效率

Pandas引入约定:

```
from pandas import Series, DataFrame
import pandas as pd
```

4.2.2 Python、Numpy和Pandas对比

`Python`

- `list`： Python自带数据类型，主要用一维，功能简单，效率
- `Dict`： Python自带数据类型，多维键值对，效率低

`Numpy`

- `ndarray`： Numpy基础数据类型，单一数据类型
- 关注数据结构/运算/维度（数据间关系）

`Pandas`

- `Series`：1维，类似带索引的1维`ndarray`
- `DataFrame`：2维，表格型数据类型，类似带行 / 列索引的2维`ndarray` 关注数据与索引的关系（数据实际应用）

从实用性、功能强弱和和可操作性比较：`list` < `ndarray` < `Series/DataFrame`

数据规整和分析工作中，`ndarry`数组作为必要补充，大部分数据尽量使用 Pandas 数据类型.

4.2.3 Pandas数据结构

`Pandas`的核心为两大数据结构，数据分析相关所有事物都是围绕着这两种结构进行的:

- `Series`
- `DataFrame` `Series`这样的数据结构用于存储一个序列的一维数据，而`DataFrame`作为更复杂的数据结构，则用于存储多维数据。

虽然这些数据结构不能解决所有的问题，但它们为大多数应用提供了有效和强大的工具。就简洁性而言，他们理解和使用起来都很简单。

两种数据结构奇特之处是讲索引对象和标签整合到自己的结构中，使得两种数据结构都具有很强的可操作性.

Series

> `Series`是一种类似于一维数组的对象，它由一组数据(各种`Numpy`数据类型)以及一组与之相关的数据标签(即索引)组成。仅由一组数据即可产生最简单的`Series`.
>
> **Series示例**
>
> `Series`的字符串表现形式为：索引在左边，值在右边。由于我们没有为数据指定索引，于是会自动创建一个`0~N-1`的整数索引。我们可以通过`Series`的`index`和`values`属性获取索引和值的数组表现形式。

```
import pandas as pd
pd.Series([i for i in range(1,6)])
```

输出结果为：

```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

Series基本操作

```
import pandas as pd
s1 = pd.Series([i for i in range(1,6)])
```

- 访问所有的索引

```
s1.index
RangeIndex(start=0, stop=5, step=1)
```

- 访问所有的值

```
s1.values
array([1, 2, 3, 4, 5])
```

- 传入 `ndarray`

```
s2 = pd.Series(np.arange(10))
s2
0    0
1    1
2    2
3    3
4    4
5    5
6    6
7    7
8    8
9    9
dtype: int64
```

- 传入字典

```
s3 = pd.Series({'1': 1, '2': 2, '3': 3})
s3
1    1
2    2
3    3
dtype: int64
s3.index
Index(['1', '2', '3'], dtype='object')
s3.values
array([1, 2, 3])
```

- 指定 index

```
s4 = pd.Series([1, 2, 3, 4], index=['A', 'B', 'C',
'D'])
s4
A    1
B    2
C    3
D    4
dtype: int64
s4.values
array([1, 2, 3, 4])
s4.index
Index(['A', 'B', 'C', 'D'], dtype='object')
```

- Series 的访问

```
s4['A']
1
```

- Series 的筛选

```
s4[s4>2]
C    3
D    4
dtype: int64
```

- 转化为字典

```
s4.to_dict()
{'A': 1, 'B': 2, 'C': 3, 'D': 4}
# 如果只传入一个字典，则结果Series中的索引就是原字典的键（有序排列）。你可以传入排好序的字典的键以改变顺序：
s5 = pd.Series(s4.to_dict())
s5
A    1
B    2
C    3
D    4
dtype: int64
index_1 = ['A', 'B', 'C','D','E']
s6 = pd.Series(s5, index = index_1)
s6 #由于E这个索引没有对应的值，即为NaN（即“非数字”（not a number），在pandas中，它用于表示缺失或NA值
A    1.0
B    2.0
C    3.0
D    4.0
E    NaN
dtype: float64
```

- 我将使用缺失（missing）或NA表示缺失数据。`pandas`的`isnull`和`notnull`函数可用于检测缺失数据：

```
pd.isnull(s6)
A    False
B    False
C    False
```

```
D    False
E     True
dtype: bool
# Series也有类似的实例方法，面向对象调用
s6.isnull()
A    False
B    False
C    False
D    False
E     True
dtype: bool
pd.notnull(s6)
A     True
B     True
C     True
D     True
E    False
dtype: bool
```

- `Series`对象本身及其索引都有一个`name`属性，该属性跟`pandas`其他的关键功能关系非常密切：

```
s6.name = 'DEMO1'
s6
A    1.0
B    2.0
C    3.0
D    4.0
E    NaN
```

```
Name: DEMO1, dtype: float64
s6.index.name = 'demo_index'
s6
demo_index
A    1.0
B    2.0
C    3.0
D    4.0
E    NaN
Name: DEMO1, dtype: float64
```

- `Series`的索引可以通过赋值的方式就地修改：

```
s6.index
Index(['A', 'B', 'C', 'D', 'E'], dtype='object',
name='demo_index')
import string
up_index = [i for i in
string.ascii_lowercase[21:26]]
up_index
['v', 'w', 'x', 'y', 'z']
s6.index = up_index
s6
v    1.0
w    2.0
x    3.0
y    4.0
z    NaN
Name: DEMO1, dtype: float64
```

Series操作

1. 选取操作

`Series`对象支持通过以下方式查询数据:

> 1. 位置下标 2. 标签索引3. 切片索引 4. 布尔型索引

```
from pandas import Series
import pandas as pd
series1 = Series([10, 20, 30, 40],
index=list('abcd'))
# 通过位置查询
series1[2]
30
# 通过标签索引查询
series1['b']
20
# 查询多个元素
series1[[0, 2, 3]]
series1[['a', 'c', 'd']]
a    10
c    30
d    40
dtype: int64
# 切片
series1[1:3]
series1['b':'d']
b    20
```

```
c    30
d    40
dtype: int64
# 布尔索引
series1[series1 > 20]
c    30
d    40
dtype: int64
```

- 删除操作

我们队对Series中的元素执行删除，主要用到Series.drop函数和pop函数.drop删除数据之后，返回删除后的副本.Series.pop在删除源数据的元素

```
series1 = Series([10, 20, 30, 40],
index=list('abcd'))
series1
a    10
b    20
c    30
d    40
dtype: int64
# 删除元素返回副本
series1.drop('c')
a    10
b    20
d    40
dtype: int64
```

```
series1.drop(['a', 'd'])
b    20
c    30
dtype: int64
# 删除源数据中的元素
series1.pop('d')
40
```

- 插入操作

```
series1 = Series([10, 20, 30, 40],
index=list('abcd'))
series2 = Series([100, 200], index=['g', 'h'])
# 新增一个标签索引值为f，值为100的元素
series1['f'] = 100
series1.append(series2)
a     10
b     20
c     30
d     40
f    100
g    100
h    200
dtype: int64
```

- `Series`运算

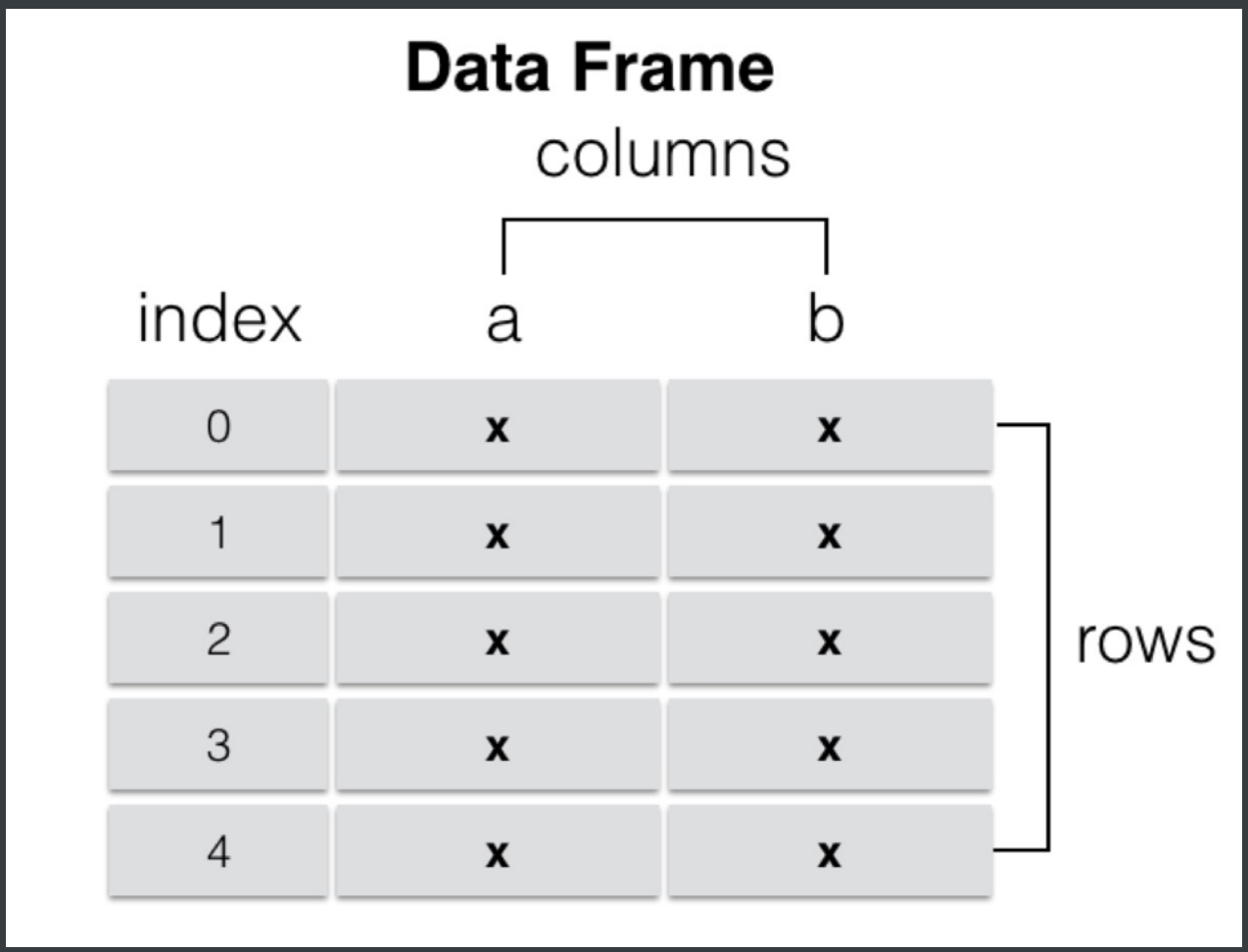
```
import numpy as np
```

```
series1 = Series([10, 20, 30, 40],
index=list('abcd'))
series2 = Series([1, 2, 3, 4], index=list('abce'))
series1 * 2
a    20
b    40
c    60
d    80
dtype: int64
# 两个series运算，会自动对齐运算
series1 * series2
a    10.0
b    40.0
c    90.0
d     NaN
e     NaN
dtype: float64
# 使用Numpy函数
np.sum(series1)
100
# 不对齐运算,将series当做numpy运算
np.add(series1, series2)
a    11
b    22
c    33
d    44
dtype: int64
np.greater(series1, series2)
a    True
```

```
b     True
c     True
d     True
dtype: bool
```

DataFrame

`DataFrame` 这种列表式的数据结构和 `Excel工` 作表非常类似，其设计初衷是将 `Series` 的使用场景由一维扩展到多维. `DataFrame` 由按一定顺序的多列数据组成，各列的数据类型可以有所不同(数值、字符串、布尔值).



> Series 对象的 Index 数组存放有每个元素的标签，而 DataFrame 对象有所不同，它有两个索引数组。第一个索引数组与行有关，它与 Series 的索引数组极为相似。 每个标签与标签所在行的所有元素相关联。而第二个数组包含一系列标签，每个标签与一列数据相关联. DataFrame 还可以理解为一个由 Series 组成的字典，其中每一列的列名为字典的键，每一个 Series 作为字典的值.

创建 DataFrame 对象最常用的方法是传递一个 dict 对象给 DataFrame 构造函数。 DataFrame以dict 的键作为列名，每个键都有一个数组作为值. 也可以通过嵌套的 dict 、 list 来创建 DataFrame 对象.

DataFrame的创建

- 通过字典来创建DataFrame

```
from pandas import DataFrame, Series
import pandas as pd

persons = {
    'name': ['小睿', '小丽', '小明', '小红'],
    'age': [19, 18, 18, 17],
    'sex': ['男', '男', '女', '男'],
}
# 字典的key作为列索引
data_frame1 = DataFrame(persons)
data_frame1
```

	name	age	sex
0	小睿	19	男
1	小丽	18	男
2	小明	18	女
3	小红	17	男

```
contries = {
    '中国': {'2013': 10, '2014': 20, '2015': 30},
    '阿富汗': {'2013': 12, '2014': 25, '2015': 33},
    '新加坡': {'2013': 11, '2014': 22, '2015': 38},
    '柬埔寨': {'2013': 18, '2014': 16, '2015': 27},
}
# 外层key做列索引，内层key做行索引
data_frame3 = DataFrame(contries)
data_frame3
```

	中国	阿富汗	新加坡	柬埔寨
2013	10	12	11	18
2014	20	25	22	16
2015	30	33	38	27

- 通过二维Numpy数组创建DataFrame

```
import numpy as np

ndarray1 = np.random.randint(1, 10, (3, 4))

# 使用默认行索引和列索引
data_frame2 = DataFrame(ndarray1)
data_frame2
```

	0	1	2	3
0	2	3	5	2
1	4	6	3	5
2	9	8	6	8

- 通过列表创建DataFrame

```
data_frame3 = DataFrame([1, 2, 3, 4, 5])
data_frame4 = DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
data_frame3
```

	0
0	1
1	2
2	3
3	4
4	5

DataFrame索引

```
# 打印DataFrame行索引
contries = {
    '俄罗斯': {'2013': 10, '2014': 20, '2015': 30},
    '阿富汗': {'2013': 12, '2014': 25, '2015': 33},
    '新加坡': {'2013': 11, '2014': 22, '2015': 38},
    '柬埔寨': {'2013': 18, '2014': 16, '2015': 27},
}
# 外层key做列索引，内层key做行索引
data_frame5 = DataFrame(contries)
data_frame5
```

	俄罗斯	阿富汗	新加坡	柬埔寨
aaa	10	12	11	18
bbb	20	25	22	16
ccc	30	33	38	27

```
# 行索引
data_frame5.index
Index(['2013', '2014', '2015'], dtype='object')
# 列索引
data_frame5.columns
Index(['俄罗斯', '阿富汗', '新加坡', '柬埔寨'],
dtype='object')
# 修改行索引
data_frame5.index = ['aaa', 'bbb', 'ccc']
data_frame5
```

```
# 修改列索引
data_frame5.columns = ['一', '二', '三', '四']
data_frame5
```

	一	二	三	四
aaa	10	12	11	18
bbb	20	25	22	16
ccc	30	33	38	27

Dataframe的操作

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np

contries = {
    '中国': {'2013': 10, '2014': 20, '2015': 30},
    '阿富汗': {'2013': 12, '2014': 25, '2015': 33},
    '新加坡': {'2013': 11, '2014': 22, '2015': 38},
    '柬埔寨': {'2013': 18, '2014': 16, '2015': 27},
}
# 外层key做列索引，内层key做行索引
data_frame1 = DataFrame(contries)
data_frame1
```

	中国	阿富汗	新加坡	柬埔寨
2013	10	12	11	18
2014	20	25	22	16
2015	30	33	38	27

- 获取元素

```
# 获取列数据
data_frame1['阿富汗']
2013    12
2014    25
2015    33
Name: 阿富汗, dtype: int64
# 获取指定多列数据
data_frame1[['柬埔寨', '中国', '阿富汗']]
```

	柬埔寨	中国	阿富汗
2013	18	10	12
2014	16	20	25
2015	27	30	33

对于DataFrame的行的标签索引，我引入了特殊的标签运算符loc和iloc。它们可以让你用类似NumPy的标记，使用轴标签（loc）或整数索引（iloc），从DataFrame选择行和列的子集。

```
# 获得列切片数据
data_frame1.iloc[:, 1: 3]
```

	柬埔寨	中国	阿富汗
2013	18	10	12
2014	16	20	25
2015	27	30	33

```
# 获取行数据，loc里面可以放字符串标签和布尔值、数组
data_frame1.loc['2014']
中国      20
阿富汗     25
新加坡     22
柬埔寨     16
Name: 2014, dtype: int64
# 获取多行数据
data_frame1.loc[['2013', '2014']]
```

	中国	阿富汗	新加坡	柬埔寨
2013	10	12	11	18
2014	20	25	22	16

```
data_frame1.loc[['2013', '2014'], ['阿富汗','柬埔寨']]
```

	阿富汗	柬埔寨
2013	12	18
2014	25	16

```
# 查找中国大于15的行
data_frame1.loc[data_frame1.阿富汗 > 15]
```

```
# 查找中国大于15的行，保留中国  阿富汗 两列
data_frame1.loc[data_frame1.中国 > 12, ['阿富汗',
'中国']]
```

```
# 位置下标获取行数据
data_frame1.iloc[0]
中国      10
阿富汗     12
新加坡     11
柬埔寨     18
Name: 2013, dtype: int64
# 切片获得区间数据，以下两种效果一样
data_frame1.iloc[1:]
data_frame1[1:]
```

```
# 数组切片索引
data_frame1.iloc[1:, [0, 1]]
```

```
# 数组切片索引
data_frame1.iloc[1:, 1:]
```

Dataframe插入数据

```
contries = {
    '中国': {'2013': 10, '2014': 20, '2015': 30},
    '阿富汗': {'2013': 12, '2014': 25, '2015': 33},
    '新加坡': {'2013': 11, '2014': 22, '2015': 38},
    '柬埔寨': {'2013': 18, '2014': 16, '2015': 27},
}
# 外层key做列索引，内层key做行索引
data_frame2 = DataFrame(contries)
```

	中国	阿富汗	新加坡	柬埔寨
2013	10	12	11	18
2014	20	25	22	16
2015	30	33	38	27

```
# 插入一行数据
```

```
series1 = Series([100, 200, 300, 400], index=['阿富汗', '柬埔寨', '新加坡', '中国'], name='2019')
series1
阿富汗    100
柬埔寨    200
新加坡    300
中国      400
Name: 2019, dtype: int64
data_frame2.loc['2016'] = 66
data_frame2.loc['2017'] = [23, 24, 25, 26]
data_frame2.loc['2018'] = series1
# append调用之后会产生数据副本
data_frame3 = data_frame2.append(series1)
data_frame3
```

	中国	阿富汗	新加坡	柬埔寨
2013	10	12	11	18
2014	20	25	22	16
2015	30	33	38	27
2016	66	66	66	66
2017	23	24	25	26
2018	400	100	300	200
2019	400	100	300	200

```
# 在最后一列增加一列新数据
data_frame3['法兰西'] = Series(np.arange(7), index=
[str(x) for x in range(2013, 2020)])

# 在指定位置增加一列
data_frame3.insert(2, '马来西亚',
Series(np.random.randint(0, 7, 7), index=[str(x)
for x in range(2013, 2020)]))
data_frame3.insert(4, '印度', np.random.randint(0,
7, 7))
data_frame3
```

Dataframe中删除数据

```
contries = {
    '中国': {'2013': 10, '2014': 20, '2015': 30},
    '阿富汗': {'2013': 12, '2014': 25, '2015': 33},
    '新加坡': {'2013': 11, '2014': 22, '2015': 38},
    '柬埔寨': {'2013': 18, '2014': 16, '2015': 27},
    '印度': {'2013': 18, '2014': 16, '2015': 27},
    '锡金': {'2013': 18, '2014': 16, '2015': 27},
    '法兰西': {'2013': 18, '2014': 16, '2015': 27},
}
# 外层key做列索引，内层key做行索引
data_frame1 = DataFrame(contries)
data_frame1
```

```
# 删除一列数据
del data_frame1['法兰西']
data_frame1
```

```
data_frame1.pop('中国')
2013    10
2014    20
2015    30
Name: 中国, dtype: int64
data_frame1
```

```
# 使用drop方法删除多列数据
data_frame2 = data_frame1.drop('锡金', axis=1)  # 0跨行 1跨列
data_frame2 = data_frame1.drop(['新加坡', '印度'], axis=1)  # 0跨行 1跨列
# data_frame2 = data_frame1.drop(['2014', '2015'], axis=0)  # 删除行
data_frame2
```

4.2.4 DataFrame算数运算和数据对齐

pandas 能将两个数据结构的索引对齐，这可能是与 pandas 数据结构索引有关的最强大的功能。这一点尤其体现在数据结构之间的算数运算上.参与运算的两个数据结构，其索引顺序可能不一致，而且有的索引项可能只存在一个数据结构中.

- Series 之间运算，遵循索引对齐

```
from pandas import Series, DataFrame
import pandas as pd

series1 = Series([10, 20, 30, 40],
index=list('abcd'))
series2 = Series([20, 30, 40, 50],
index=list('bcde'))
# 如果索引并不重叠，则使用NaN
series1 + series2
a     NaN
b    40.0
c    60.0
d    80.0
e     NaN
dtype: float64
```

- `DataFrame`运算会对齐行和列索引

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np

data_frame1 = DataFrame(
    np.arange(12).reshape((3, 4)),
    index=['fir', 'sec', 'thr'],
    columns=['a', 'b', 'c', 'd']
)

data_frame2 = DataFrame(
```

```
    np.arange(8, 20).reshape((4, 3)),
    index=['fir', 'sec', 'thr', 'for'],
    columns=['a', 'b', 'c']
)
data_frame1
```

```
data_frame2
```

```
# data_frame1 + data_frame2
# 在算数方法中填充值,当某个标签在另一个对象中找不到时填充一个值。
data_frame1.add(data_frame2, fill_value=0)
```

- DataFrame和Series之间的运算

```
import numpy as np
data_frame2 = DataFrame(np.arange(9).reshape((3,
3)),
                        index=['a', 'b', 'c'],
                        columns=['num1', 'num2',
'num3'])

series2 = Series([10, 20, 30, 40],
                 index=['num1', 'num2', 'num3',
'num4'])
data_frame2
```

```
series2
num1    10
num2    20
num3    30
num4    40
dtype: int64
# DataFrame每一行都加上series
data_frame2 + series2
```

```
# DataFrame每一列加上series
series3 = Series([10, 20, 30, 40], index=['a',
'b', 'c', 'd'])
data_frame2.add(series3, axis=0)
```

4.2.5 pandas中函数应用apply

pandas 库以 Numpy 为基础，并对它的很多功能进行了扩展,用来操作新的数据结构 Series 和 DataFrame .通用函数(ufunc, universal function)就是扩展得到的功能，这类函数能够对数据结构中的元素进行操作，特别有用.除了通用函数，用户还可以自定义函数。

数组中的大多数函数对DataFrame依然有效，因此没有必须要使用apply函数.

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np

data_frame1 = DataFrame(np.arange(-8,
8).reshape((4, -1)))
data_frame1
```

	0	1	2	3
0	-8	-7	-6	-5
1	-4	-3	-2	-1
2	0	1	2	3
3	4	5	6	7

```
# 计算每一个元素的平方
np.square(data_frame1)
```

- 将函数应用到各行或各列所形成的一维数组上

```
def apply_function(x):
    return np.sum(x)
# 将每一列数据应用到apply_function上
data_frame1.apply(apply_function)
0   -8
1   -4
2    0
3    4
dtype: int64
# 将每一行数据应用到apply_function上
data_frame1.apply(apply_function, axis=1)
0   -26
1   -10
2     6
```

```
3    22
dtype: int64
# applymap可以将每一个元素应用到函数中
data_frame1.applymap(lambda x: x + 100)
```

Pandas排序

根据条件对数据集进行排序也是一种重要的内置运算. 要对行或列索引进行排序，可使用sort_index方法，要根据元素值来排序，可使用sort_values.

```
def sort_index(self, axis=0, ascending=True,
inplace=False):
    pass

def sort_values(self, by, axis=0, ascending=True,
inplace=False):
    pass
```

- axis: 按行还是按列排序.
- ascending: 升序还是降序,默认True是升序排列,False降序排序.
- inplace: 默认False, 返回排序副本. True表示直接修改序列本身
- 对Series排序

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np
```

```
series1 = Series(np.random.randint(1, 100, 4),
index=['d', 'a', 'c', 'b'])
series1
d    81
a    85
c    33
b    44
dtype: int64
# 根据索引升序排列
series1.sort_index()
a    85
b    44
c    33
d    81
dtype: int64
# 根据索引降序排列
series1.sort_index(ascending=False, inplace=True)
series1
d    81
c    33
b    44
a    85
dtype: int64
# 按照值对Series升序排列
series1.sort_values()
c    33
b    44
d    81
a    85
```

```
dtype: int64
# 按照值对Series降序排列
series1.sort_values(ascending=False)
a    85
d    81
b    44
c    33
dtype: int64
```

- 对DataFrame排序

```
data_frame1 = DataFrame(
    np.arange(16).reshape((4, 4)),
    index=['b', 'a', 'd', 'c'],
    columns=[4, 2, 3, 1]
)
data_frame1
```

```
# 根据行索引进行排序
data_frame1.sort_index()
# 根据行索引进行降序排列
data_frame1.sort_index(ascending=False)
```

```
# 根据列索引排序
data_frame1.sort_index(axis=1)
# 根据列索引降序排列
data_frame1.sort_index(ascending=False, axis=1,
inplace=True)
data_frame1
```

```
# 按照值对行进行升序排列
data_frame1.sort_values(by=1)
# 按照值对行进行降序排列
data_frame1.sort_values(by=1, ascending=False)
```

```
# 按照值对指定列进行升序排列
data_frame1.sort_values(by='a', axis=1)
# 按照值对指定列进行降序排列
data_frame1.sort_values(by='a', axis=1,
ascending=False)
```

等级索引和分级

等级索引是pandas的一个重要功能，单条轴可以有多级索引. 你可以像操作两位结构那样处理多维数据.

```
from pandas import Series, DataFrame
```

```
import pandas as pd
import numpy as np

areas_economy = Series(
    np.arange(10, 100, 10),
    index=[
        ['北京市', '北京市', '北京市', '河北省', '河北省', '河北省', '山东省', '山东省', '山东省'],
        ['信息业', '制造业', '服务业', '信息业', '制造业', '服务业', '信息业', '制造业', '服务业']])

areas_economy
北京市  信息业    10
     制造业    20
     服务业    30
河北省  信息业    40
     制造业    50
     服务业    60
山东省  信息业    70
     制造业    80
     服务业    90
dtype: int64
```

- 层级索引取值和索引sort_index排序

```
# 1. 获得一个省的信息
areas_economy['山东省']
信息业    70
制造业    80
```

```
服务业    90
dtype: int64
# 2. 获得多个省信息
areas_economy[['河北省', '山东省']]
河北省  信息业    40
      制造业    50
      服务业    60
山东省  信息业    70
      制造业    80
      服务业    90
dtype: int64
# 3. 获得所有省的制造业信息
areas_economy[:, '制造业']
# 获得多个省制造业信息错误写法
# areas_economy[['河北省', '山东省'], '制造业']
北京市    20
河北省    50
山东省    80
dtype: int64
# 4. 获得多个省制造业信息
areas_economy.loc[['河北省', '山东省'], '制造业']
河北省  制造业    50
山东省  制造业    80
dtype: int64
# 5. 获得多个省制造业、服务业信息
areas_economy.loc[['河北省', '山东省'], ['制造业',
'服务业']]
河北省  制造业    50
      服务业    60
```

```
山东省  制造业     80
        服务业     90
dtype: int64
# 需要对最外层排序之后才可执行切片索引
areas_economy.sort_index()['北京市':'山东省']
北京市  信息业     10
        制造业     20
        服务业     30
山东省  信息业     70
        制造业     80
        服务业     90
dtype: int64
# 交换索引
areas_economy.swaplevel()
信息业  北京市     10
制造业  北京市     20
服务业  北京市     30
信息业  河北省     40
制造业  河北省     50
服务业  河北省     60
信息业  山东省     70
制造业  山东省     80
服务业  山东省     90
dtype: int64
# 对索引进行排序，可通过level=0或1指定根据那层索引排序
areas_economy.sort_index(level=0)
北京市  信息业     10
        制造业     20
        服务业     30
```

```
山东省  信息业    70
      制造业    80
      服务业    90
河北省  信息业    40
      制造业    50
      服务业    60
dtype: int64
```

- 按层级统计数据

```
# 计算北京市经济总量
areas_economy['北京市'].sum()
# 计算山东省三大行业的平均经济总量
areas_economy['山东省'].mean()
80.0
# 统计每个行业的经济总量
areas_economy.sum(level=1)
信息业    120
制造业    150
服务业    180
dtype: int64
data_frame1 = DataFrame(
    np.arange(0, 360, 10).reshape((6, 6)),
    index=[
        ['北京市', '北京市', '河北省', '河北省', '河南省', '河南省'],
        ['昌平区', '海淀区', '石家庄', '张家口', '驻马店', '平顶山'],
    ],
```

```
    columns=[
        ['轻工业', '轻工业', '重工业', '重工业', '服务
业', '服务业'],
        ['纺织业', '食品业', '冶金业', '采煤业', '教育
业', '游戏业'],
    ]
)

data_frame1
```

```
data_frame1['重工业']
```

```
data_frame1.loc['河北省']
data_frame1.loc['河北省']['重工业']
```

4.2.6 处理缺失数据

缺失数据(missing data)在大部分数据分析应用中都很常见。补上缺失值非常用，它们在数据结构中用NaN来表示，便于识别.

处理缺失值一般有三种方式

- 直接丢弃数据中的包含NaN的行或列.

```
from pandas import DataFrame, Series
```

```
import pandas as pd
import numpy as np

data_frame1 = DataFrame([
    [10, 20, 30, 40, 50],
    [11, 21, np.nan, 41, np.nan],
    [12, 22, 32, 42, 52],
    [np.nan, 66, np.nan, np.nan, np.nan],
    [np.nan, 24, np.nan, 44, 54],
])
data_frame1
# 默认丢弃包含NaN的行
data_frame1.dropna()
```

```
# 丢弃包含NaN的列
data_frame1.dropna(axis=1)
```

```
# 如果只是想删除所有元素是NaN的行或列, 需要参数how='all'
data_frame1.dropna(how='all', axis=1)
```

- 在选取数据的时候过滤掉包含NaN的行或列.

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np

data_frame1 = DataFrame([
    [10, 20, 30, 40, 50],
    [11, 21, np.nan, 41, np.nan],
    [12, 22, 32, 42, 52],
    [np.nan, 66, np.nan, np.nan, np.nan],
    [np.nan, 24, np.nan, 44, 54],
])
data_frame1
```

```
# 获得第一列不为NaN的元素
data_frame1[0][data_frame1[0].notnull()]
0    10.0
1    11.0
2    12.0
Name: 0, dtype: float64
# 获得第三行不为NaN的元素
data_frame1.loc[3][~(data_frame1.loc[3].isnull())]
data_frame1.loc[3][data_frame1.loc[3].notnull()]
1    66.0
Name: 3, dtype: float64
```

- 填充NaN值

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np

data_frame1 = DataFrame([
    [10, 20, 30, 40, 50],
    [11, 21, np.nan, 41, np.nan],
    [12, 22, 32, 42, 52],
    [np.nan, 66, np.nan, np.nan, np.nan],
    [np.nan, 24, np.nan, 44, 54],
])
data_frame1
```

```
data_frame1.fillna(666)
```

```
# 每一列替换为不同的值，目前不支持对按行填充不同值的功能.
data_frame1.fillna({0: 100, 1: 200, 2:300, 3: 400,
4: 500})
```

4.2.7 数据准备

我们从文件或数据库等数据源获取数据之后，将数据转换为DataFrame格式后，我们就可以对数据进行处理了。数据处理的目的是准备数据，便于我们后续分析数据。

- 我们下面将会深入学习pandas库在数据处理阶段的功能. 数据处理分为三个阶段:

1.数据准备 2.数据转换 3.数据聚合

- 开始处理数据工作之前，需要先行准备好数据，把数据组装成便于用pandas库的各种工具处理的数据结构。数据准备阶段包括以下步骤:

1.数据加载 2. 组装:合并(merging)拼接(concatenation)组合(combine)3.变形(轴向旋转)

数据加载就是从文件、数据库等数据源中讲待处理的数据加载到程序中，并转换为DataFrame等结构。数据可能来自不同的数据源，有着不同的格式，需要将其归并为一个DataFrame, 然后才能做进一步的处理.

数据组装

对于存储在pandas对象中的各种数据，组装的方法有以下几种:

- 合并--pandas.merge()函数根据一个或多个键连接多行.
- 拼接--pandas.concat()函数按照轴把多个对象拼接起来.
- 结合--pandas.DataFrame.combine_first()函数从另外一个数据结构获取数据，连接重合的数据，填充缺失值.

合并(merge)

> 对于合并操作，熟悉SQL的读者可以将其理解为JOIN操作，它使用一个或多个键把多行数据结合在一起.

Merge通过索引来合并数组

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np


data_frame1 = DataFrame({
    'name': ['John', 'Edward', 'Smith', 'Obama',
'Clinton' ],
    'ages1': [18, 16, 15, 14, 19]
})

data_frame2 = DataFrame({
    'name': ['John', 'Edward', 'Polly', 'Obama' ],
    'ages2': [19, 16, 15, 14]
})

data_frame3 = DataFrame({
    'other_name': ['John', 'Edward', 'Smith',
'Obama', 'Bush' ],
    'other_ages': [18, 16, 15, 14, 20]
})
print(data_frame1)
print('-'*10)
print(data_frame2)
```

```
print('-'*10)
print(data_frame3)
      name  ages1
0     John     18
1   Edward     16
2    Smith     15
3    Obama     14
4  Clinton     19
----------
     name  ages2
0    John     19
1  Edward     16
2   Polly     15
3   Obama     14
----------
  other_name  other_ages
0       John          18
1     Edward          16
2      Smith          15
3      Obama          14
4       Bush          20
# 默认根据相同列名作为键连接两个DataFrame，并且内连接方式(inner，取两边都有的数据，交集)
# 那么请问，当我将data_frame2的列名ages2改为ages1时，结果是多少?
pd.merge(data_frame1, data_frame2, on='name')
```

```
# 可修改连接方式为外连接(outer),取两遍都有数组，缺失数据使用NaN表示
pd.merge(data_frame1, data_frame2, how='outer')
```

```
# 当两个DataFrame没有公共相同的列名的时候，那么合并就会报错.
# MergeError: No common columns to perform merge on
# 此时我们可指定DataFrame用那一列名来连接另一个DataFrame的那个列名
# 使用参数 left_on 和 right_on
pd.merge(data_frame1, data_frame3, left_on='name', right_on='other_name')
```

```
# 还可以在此基础上使用外连接
pd.merge(data_frame1, data_frame3, left_on='name', right_on='other_name', how='outer')
```

```
# 除了内外连接，还有左连接(left)和右连接(right)
# 左连接以左DataFrame为准，右DataFrame不存在的列用NaN代替.
pd.merge(data_frame1, data_frame2, how='left')
```

```
# 左连接以右DataFrame为准，左DataFrame不存在的列用NaN代替.
pd.merge(data_frame1, data_frame2, how='right')
```

```
# 有时候，DataFrame中连接的键在索引(index)中。
# 在这种情况下传入left_index=True, right_index=True
# 说明索引被应用于连接键.
pd.merge(data_frame1, data_frame2,
left_index=True, right_index=True, suffixes=['_a', '_b'])
```

拼接(concat)

concat函数可以将DataFrame、Series根据不同的轴作简单的融合

pd.concat(objs, axis=0, join='outer')

> 参数说明:objs: series，dataframe构成的序列lsit axis： 需要合并链接的轴，0是行，1是列 join：连接的方式 inner，或者outer

```
from pandas import DataFrame, Series
import pandas as pd
```

```
df1 = DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                 'B': ['B0', 'B1', 'B2', 'B3'],
                 'C': ['C0', 'C1', 'C2', 'C3'],
                 'D': ['D0', 'D1', 'D2',
'D3']},
                 index=[0, 1, 2, 3])

df2 = DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                 'B': ['B4', 'B5', 'B6', 'B7'],
                 'C': ['C4', 'C5', 'C6', 'C7'],
                 'D': ['D4', 'D5', 'D6',
'D7']},
                  index=[4, 5, 6, 7])

df3 = DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                 'B': ['B8', 'B9', 'B10',
'B11'],
                 'C': ['C8', 'C9', 'C10',
'C11'],
                 'D': ['D8', 'D9', 'D10',
'D11']},
                 index=[8, 9, 10, 11])

df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
                 'D': ['D2', 'D3', 'D6', 'D7'],
                 'F': ['F2', 'F3', 'F6',
'F7']},
                 index=[2, 3, 6, 7])
```

```
result = pd.concat([df1, df2, df3])
# 默认是将后者DataFrame向纵方向拼接.
result
```

```
# 上面拼接完成之后，我们不清楚在Result中哪些数据是
data_frame1
# 哪些数据是data_frame1，我们可以通过keys为其增加等级索
引
result = pd.concat([df1, df2, df3], keys=['x',
'y', 'z'])
result
```

下面我们看看如何横向拼接两个DataFrame:

- 横向拼接，我们只需要将设置axis=1.

```
result = pd.concat([df1, df4], axis=1)
result
```

我们下面看看join如何用, join有两个可选项, 默认是outer(两个DataFrame并集), 也可设置为inner(两个DataFrame交集).

```
result = pd.concat([df1, df4], axis=1,
join='inner')
result
```

组合(combine)

假如我们有索引全部或部分重叠的两个数据集. 我们可能希望用一个数据集中的元素去补充另外一个数据集中的元素.例如:

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np

series1 = Series([np.nan, 10, 20, np.nan, 30],
index=list('abcde'))
series2 = Series([100, 11, 22, 200, np.nan],
index=list('abcde'))

series1

#    a     NaN
#    b    10.0
#    c    20.0
#    d     NaN
#    e    30.0
#    dtype: float64

series2

#    a    100.0
#    b     11.0
#    c     22.0
#    d    200.0
#    e      NaN
#    dtype: float64
```

```
# 如果series1中的元素为NaN，那么用series2中的元素来补充
# 如果series2中的元素在series1中不存在，则在series1中新增元素
series1.combine_first(series2)

#    a    100.0
#    b     10.0
#    c     20.0
#    d    200.0
#    e     30.0
#    dtype: float64
a    100.0
b     10.0
c     20.0
d    200.0
e     30.0
dtype: float64
```

轴向旋转

我们除了要将来自不同数据源的进行整合统一之外，另外一种常用操作成为轴向旋转(pivoting). 轴向旋转操作我们主要用到两个函数:

1.入栈(stacking):旋转数据结构，把列转换为行. 2.出栈(unstacking):把行转换为列

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np

data_frame = DataFrame(
    np.arange(9).reshape((3, 3)),
    index=list('abc'),
    columns=list('efg')
)

data_frame
```

```
# 将data_frame的列['e', 'f', 'g']转换为行
data_frame.stack()
a  e    0
   f    1
   g    2
b  e    3
   f    4
   g    5
c  e    6
   f    7
   g    8
dtype: int64
# 将行转换为列
data_frame.stack().unstack()
```

```
# 将行转换为列
# 0表示将['a', 'b', 'c']转换为列
# 1表示将['e', 'f', 'g']转换为列
data_frame.stack().unstack(0)
```

4.2.8 数据转换

我们已经学习了如何准备数据以便于进行数据分析. 这个过程体现在重组DataFrame中的数据上.现在我们进行第二步数据转换.

在数据转换过程中，有些操作会涉及重复或无效元素，可能需要将其删除或者替换为别的元素.

删除重复行

数据处理的最后一步是删除多余的列和行.前面我们已经学习了del、DataFrame.drop()删除行列.由于多种原因，DataFrame对象可能包含重复的行.在大型的DataFrame中，检测重复的行可能遇到各种问题。pandas为此提供了多种工具，便于分析大型数据中的重复数据.

这里我们主要学习连个用于检测和删除重复行的函数

1.duplicated函数,用于检测重复行，返回元素为Bool类型的Series对象. 2.drop_duplicates, 用于从DataFrame对象中删除重复的行.

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np

data_frame = DataFrame({
    'name': ['John', 'John', 'Edward', 'Smith',
'Edward', 'John'],
    'age': [19, 19, 20, 21, 20, 19]
})

data_frame
```

```
# 检测是否存在重复行
data_frame.duplicated()
0    False
1     True
2    False
3    False
4     True
5     True
dtype: bool
# 删除重复行
data_frame.drop_duplicates()
```

映射相关

pandas提供了几个利用映射关系来实现某些操作的函数。映射关系无非就是创建一个映射关系的列表，把元素跟一个特定的标签或者字符串绑定起来.

我们主要学习以下三个映射函数:1.replace(): 替换元素 2. rename(): 替换索引 3.map(): 新建一列.

替换元素(replace)

组装完数据结构后，里面通常会有些元素不符合需求. 需要将某些文本替换为另外的文本，例如存在外语文本.

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np

data_frame1 = DataFrame({
    'name': ['xxx', 'John', 'Edward', 'Smith',
'xxx', 'yyy', np.nan],
    'age': np.arange(10, 17),
})

data_frame1
```

```
# 将DataFrame中的xxx替换为Obama, yyy替换成Polly
replace_map = {'xxx': 'Obama', 'yyy': 'Polly'}
data_frame1.replace(replace_map, inplace=True)
data_frame1
```

```
# 将DataFrame中的NaN替换成zzz
data_frame1.replace(np.nan, 'zzz')
```

重命名轴索引(rename)

pandas的rename函数可以用来替换轴的索引标签.

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np

data_frame = DataFrame(np.arange(12).reshape((4,
3)))
data_frame
```

```
re_index = {
    0: 'a',
    1: 'b',
    2: 'c',
    3: 'd'
}

re_column = {
    0: 'fir',
    1: 'sec',
```

```
    2: 'thi'
}

# 修改索引标签
# 也可以指定哪些标签需要修改，不需要每次都指定
data_frame.rename(index=re_index,
columns=re_column, inplace=True)
data_frame
```

使用映射添加元素(map)

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np

data_frame = DataFrame({
    'name': ['John', 'Edward', 'Smith', 'Obama',
'Polly'],
    'age': np.arange(18, 23),
})

data_frame
```

```
# 我们现在有一个字典存储了学生对应的学习成绩
score = {
    'John': 89,
    'Edward': 98,
```

```
    'Smith': 85,
    'Bush': '83',
    'Obama': 78,
    'Polly': 67,
    'Donald': 77
}

# 现在我们想在data_frame中增加一列分数
data_frame['score'] =
data_frame['name'].map(score)
data_frame
```

4.2.9 分组(GroupBy)

Hanley Wickham(许多R语言包的作者)，创造了一个用于表示分组运算的术语'split-apply-combine'(拆分-应用-合并), 这个词很好描述了整个过程。 分组运算的第一个阶段, pandas对象(Series和DataFrame)中的数据会根据你所提供的一个或多个键被拆分(split)成多组. 拆分操作是在对象的特定轴上进行的. 例如，DataFrame可以在其行(axis=0)或列(axis=1)上进行分组. 然后, 讲一个函数应用(apply)到各个分组并产生一个新值. 最后，所有这些函数的执行结果会被合并(combine)到最终的结果对象中.

1. 根据某一列或多列分组

```
from pandas import DataFrame, Series
import pandas as pd
import numpy as np

data_frame = DataFrame({
    'key1': ['a', 'a', 'b', 'b', 'a'],
    'key2': ['one', 'two', 'one', 'two', 'three'],
    'data1': np.arange(5),
    'data2': np.arange(10, 15)
})

data_frame
```

```
# 按照key1对数据进行分组
data_gruop_by_key1 = data_frame.groupby('key1')

# 输出a、b两组数据
dict([group for group in data_gruop_by_key1])['a']
```

```
dict([group for group in data_gruop_by_key1])['b']
```

```
# 计算每一组的平均值
```

```
print(data_gruop_by_key1.mean())
print('-'*30)

# 每一组元素个数
print(data_gruop_by_key1.size())
print('-'*30)
          data1      data2
key1
a     1.666667  11.666667
b     2.500000  12.500000
------------------------------
key1
a    3
b    2
dtype: int64
------------------------------
# 对某一列数据进行分组
data1_gruop_by_key1 =
data_frame['data1'].groupby(data_frame['key1'])
print(data1_gruop_by_key1.mean())
key1
a    1.666667
b    2.500000
Name: data1, dtype: float64
data_frame = DataFrame({
    'key1': ['a', 'a', 'b', 'b', 'a'],
    'key2': ['one', 'one', 'one', 'two', 'three'],
    'data1': np.arange(5),
    'data2': np.arange(10, 15)
```

```
})

data_frame
```

```
# 按照key1、key2进行分组
data_gruop_by_key12 = data_frame.groupby(['key1',
'key2'])
data_gruop_by_key12.sum()
```

- 根据索引值分组

```
data_frame = DataFrame(
    np.random.randint(10, 13, (6, 6)),
    index=list('aabbcc'),
    columns=list('112233')
)

data_frame
```

```
# 根据列索引分组，并求每一组平均值
data_frame.groupby(level=0, axis=1).mean()
```

```
# 根据行索引分组，并求每一组平均值
data_frame.groupby(level=0, axis=0).mean()
```

- 手动指定索引进行分组

```
data_frame = DataFrame(
    np.arange(36).reshape((6, 6)),
    columns=list('abcdef')
)

data_frame
```

```
# 按照列进行分组
# 也就是手动指定哪些索引为一组
group_mapping = {
    'a': 'first',
    'b': 'first',
    'c': 'first',
    'd': 'second',
    'e': 'third',
    'f': 'third',
}
```

```
data_group_by_dict =
data_frame.groupby(group_mapping, axis=1)

# 输出三组信息
print(dict([x for x in data_group_by_dict])
['first'])
print('-'*13)
print(dict([x for x in data_group_by_dict])
['second'])
print('-'*13)
print(dict([x for x in data_group_by_dict])
['third'])
    a   b   c
0   0   1   2
1   6   7   8
2  12  13  14
3  18  19  20
4  24  25  26
5  30  31  32
-------------
    d
0   3
1   9
2  15
3  21
4  27
5  33
-------------
    e   f
```

```
0   4   5
1  10  11
2  16  17
3  22  23
4  28  29
5  34  35
# 对每一组进行求和计算
print(data_group_by_dict.sum())
   first  second  third
0      3       3      9
1     21       9     21
2     39      15     33
3     57      21     45
4     75      27     57
5     93      33     69
# 将行索引分组
data_frame = DataFrame(
    np.arange(36).reshape((6, 6)),
    index=list('abcdef')
)

print(data_frame)
    0   1   2   3   4   5
a   0   1   2   3   4   5
b   6   7   8   9  10  11
c  12  13  14  15  16  17
d  18  19  20  21  22  23
e  24  25  26  27  28  29
f  30  31  32  33  34  35
```

```
group_mapping = {
    'a': 'first',
    'b': 'first',
    'c': 'first',
    'd': 'second',
    'e': 'third',
    'f': 'third',
}

data_group_by_dict =
data_frame.groupby(group_mapping, axis=0)

# 输出三组信息
print(dict([x for x in data_group_by_dict])
['first'])
print('-'*25)
print(dict([x for x in data_group_by_dict])
['second'])
print('-'*25)
print(dict([x for x in data_group_by_dict])
['third'])

data_group_by_dict.sum()
    0   1   2   3   4   5
a   0   1   2   3   4   5
b   6   7   8   9  10  11
c  12  13  14  15  16  17
-------------------------
    0   1   2   3   4   5
```

```
   d  18  19  20  21  22  23
   ------------------------
      0   1   2   3   4   5
   e  24  25  26  27  28  29
   f  30  31  32  33  34  35
```

4.2.10 pandas的I/O操作

我们学习了pandas库以及它所提供的用于数据分析的基础功能，了解了DataFrame和Series是这个库的核心，数据处理、计算和分析都是围绕它们展开.

Pandas是数据分析专用库，它所关注的是数据计算和处理。从外部文件读写数据也是数据处理的重要部分，于是pandas提供了用于文件数据读写的专门的工具, 这些工具提供了将不同的数据结构写入不同格式文件的方法，而无需过多考虑所使用的技术.

这些用于读写数据文件的函数分为对称的两大类:读取函数和写入函数.

读取函数	写入函数
read_csv	to_csv
read_excel	to_excel
read_sql	to_sql
read_html	to_html
read_json	to_json

4.3 Matplotlib

Matplotlib 是一个 Python 的 2D 绘图库，通过 Matplotlib ，开发者可以仅需要几行代码，便可以生成绘图，直方图，功率谱，条形图，错误图，散点图等。

官网API https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html

- 用于创建出版质量图表的绘图工具库
- 最流行的 Python 底层绘图库，主要做数据可视化图表,名字取材于 MATLAB ， 模仿 MATLAB 构建
- 目的是为 Python 构建一个 Matlab 式的绘图接口
- pyploy 模块包含了常用的 Matplotlib API

简单线性图

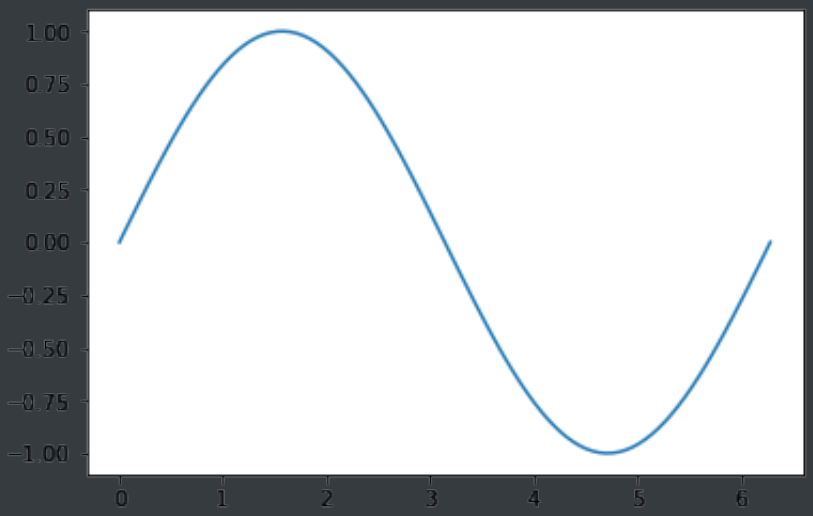
在图表的所有类型中，线性图最为简单。线性图的各个数据点由一条直线来连接. 一对对(x, y)值组成的数据点在图表中的位置取决于两条轴(x和y)的刻度范围.

如果要绘制一系列的数据点，需要创建两个Numpy数组. 首先, 创建包含x值的数组, 用作x轴. 再创建包含y值得数组，用作y轴. 完成了两个数组创建，只需要调用plot()函数绘制图像即可.

在图表的所有类型中，线性图最为简单。线性图的各个数据点由一条直线来连接. 一对对(x, y)值组成的数据点在图表中的位置取决于两条轴(x和y)的刻度范围.

如果要绘制一系列的数据点，需要创建两个Numpy数组. 首先, 创建包含x值的数组, 用作x轴. 再创建包含y值得数组，用作y轴. 完成了两个数组创建，只需要调用plot()函数绘制图像即可.

```
from matplotlib import pyplot as plt
import numpy as np
# 生成[0, 2π]之间的等间距的100个点
x = np.linspace(0, 2* np.pi,num=100)
y = np.sin(x)
plt.plot(x,y)
plt.show()
```



线条和标记节点样式: 标记字符：标记线条中的点：

- 线条颜色，color='g'
- 线条风格，linestyle='--'
- 线条粗细，linewidth=5.0

- 标记风格，marker='o'
- 标记颜色，markerfacecolor='b'
- 标记尺寸，markersize=20
- 透明度，alpha=0.5
- 线条和标记节点格式字符 如果不设置颜色，系统默认会取一个不同颜色来区别线条.

颜色字符(color)	风格(linestyle)	标记字符(mark)
r 红色	- 实线	o实心圈标记
g 绿色	-- 虚线,破折线	. 点标记
b 蓝色	-. 点划线	, 像素标记,极小的点
w 白色	: 点虚线,虚线	v 倒三角标记
c 青色	" 留空或空格,无线条	^ 上三角标记
m 洋红		> 右三角标记
y 黄色		< 左三角标记
k 黑色		* 星形标记
#00ff00 16进制		+ 十字标记

- 接下来我们绘制一个样式较全的线形图:

```
import numpy as np
import matplotlib.pyplot as plt
# 设置中文字体，否则中文会出现方框状
plt.rcParams["font.sans-serif"] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False
x = np.linspace(0, 2* np.pi,num=10)
y = np.sin(x)
# 调用绘制线性图函数plot()
plt.plot(x, y,
         color='#3589FF',  # 线的颜色
```

```
            linestyle=':',  # 线的风格
            linewidth=3,  # 线的宽度
            marker='o',   # 标记点的样式
            markerfacecolor='r',  # 标记点的颜色
            markersize=10,  # 标记点的大小
            alpha=0.7,  # 图形的透明度
            label="cos(x)"
    )
    plt.show()
```

绘制多条折线

```
    # 绘制多条折线
    x = np.linspace(0, 2* np.pi,num=20)
    y = np.sin(x)
    # 调用绘制线性图函数plot()
    plt.plot(x, y,
             color='#3589FF',  # 线的颜色
             linestyle=':',  # 线的风格
             linewidth=3,  # 线的宽度
             marker='o',   # 标记点的样式
             markerfacecolor='r',  # 标记点的颜色
             markersize=10,  # 标记点的大小
             alpha=0.7,  # 图形的透明度
             label="sin(x)"  #设置图例的label
    )

    siny = y.copy()
    cosy = np.cos(x)
```

```
plt.plot(x, cosy,
         color='y',  # 线的颜色
         linestyle='-',  # 线的风格
         linewidth=3,  # 线的宽度
         marker='*',   # 标记点的样式
         markerfacecolor='b',  # 标记点的颜色
         markersize=15,  # 标记点的大小
         alpha=0.9,  # 图形的透明度
         label="cos(x)" #设置图例的label
)

# 设置x，y轴的label
plt.xlabel('时间（s）')
plt.ylabel('电压（V）')
plt.legend()
plt.title('电压随时间变化的线性图')
# 调用show方法显式
plt.show()
```

条状图

条状图也是非常常用的一种图表类型. 条形图是统计图资料分析中最常用的图形。主要特点有：

- 能够使人们一眼看出各个各个项目数据的大小。
- 易于比较各个不同项目数据之间的差别。

```
import matplotlib.pyplot as plt
plt.rcParams["font.sans-serif"] = ['SimHei']
```

```
plt.rcParams['axes.unicode_minus'] = False
index = np.arange(5) # 返回一个有终点和起点的固定步长的排列
values1 = np.random.randint(11, 20, 5) #随机从数据中，按照一定的行数或者比例抽取数据
values2 = np.random.randint(11, 20, 5)
values3 = np.random.randint(11, 20, 5)

# bar宽度
bar_width = 0.3

# 每一个bar占0.3宽度
plt.bar(index-bar_width, values1, width=bar_width, alpha=0.7, label='社保项目1', color='b')
plt.bar(index, values2, width=bar_width, alpha=0.7, label='社保项目2', color='r')
plt.bar(index+bar_width, values3, width=bar_width, alpha=0.7, label='社保项目3', color='g')


# 显示图例
plt.legend(loc=1)

# 设置X轴、Y轴数值范围
# plt.xlim(-0.5, 5)
# plt.ylim(10, 20)
plt.axis([-0.6, 5, 10, 20])

# 设置x轴刻度标签 rotation旋转角度
```

```
plt.xticks(index, ['社保项目'+str(ix) for ix in
range(1, 6)], rotation=30)

# 设置标题
plt.title('社保项目营收', fontsize=20)
plt.xlabel('项目类型')
plt.ylabel('项目合同额（亿元）')


# 显示数值标签
# zip() 函数用于将可迭代的对象作为参数，将对象中对应的元素
打包成一个个元组，然后返回由这些元组组成的列表。
for a,b in zip(index, values1):
    plt.text(a-bar_width, b, '%.0f' % b,
ha='center', va= 'bottom', fontsize=7)

for a,b in zip(index, values2):
    plt.text(a, b, '%.0f' % b, ha='center', va=
'bottom', fontsize=7)

for a,b in zip(index, values3):
    plt.text(a+bar_width, b, '%.0f' % b,
ha='center', va= 'bottom', fontsize=7)


# 显示网格
plt.grid()

plt.show()
```

4.4 WordCloud

`WordCloud` 库，可以说是 `python` 非常优秀的词云展示第三方库。词云以词语为基本单位更加直观和艺术的展示文本。词云图，也叫文字云，是对文本中出现频率较高的“关键词”予以视觉化的展现，词云图过滤掉大量的低频低质的文本信息，使得浏览者只要一眼扫过文本就可领略文本的主旨。基于 `Python` 的词云生成类库,很好用,而且功能强大。在做统计分析的时候有着很好的应用，比较推荐。

- github:https://github.com/amueller/word_cloud
- 官方地址:https://amueller.github.io/word_cloud/

`WordCloud` 的 `API`

目前 `wordcloud` 所有功能都封装在 `WordCloud` 类中，相关 `API` 如下表所示：

API	功能
WordCloud([font_path, width, height, …])	用于生成和绘制词云对象
ImageColorGenerator(image[, default_color])	基于彩色图像的颜色生成器
random_color_func([word, font_size, …])	随机色调颜色生成

`WordCloud` 的参数

名称	类型	默认	含义

		值	
font_path	string	None	字体路径（字体为.OTF或.TTF格式）
width	int	400	词云图片宽度
height	int	200	词云图片高度
margin	int	2	词云行间距，即词与词的垂直距离
ranks_only	——	None	——
prefer_horizontal	float	0.9	值越大水平显示词越多
mask	nd-array 或 None	None	词云形状，默认矩形
scale	float	1	计算和绘图之间缩放
color_func	callable	None	为单词返回的PIL颜色
max_words	int	200	最大显示词数
min_font_size	int	4	最小字号
stopwords	set of strings 或 None	None	不显示词列表
random_state	——	None	——
background_color	color value	black	词云背景颜色

max_font_size	int 或 None	None	最大字号
font_step	int	1	云字间距，即词与词的水平距离
mode	string	RGB	——
relative_scaling	float	auto	0至1间浮点数，词频对字号的重要性
regexp	string 或 None	None	正则表达式
collocations	bool	True	是否包括两个单词的搭配（字母组合）
colormap	string 或 matplotlib colormap	None	颜色图
normalize_plurals	bool	True	是否从词中删除结尾的“s”
contour_width	float	0	词云形状边宽宽度
contour_color	color value	black	词云形状边宽颜色
repeat	bool	False	是否重复词
include_numbers	bool	False	是否包含数字
min_word_length	int	0	一个词必须包含的最小字母数

`WordCloud` 的方法

名称	功能
fit_words(self, frequencies)	根据单词和频率创建一个wordcloud
generate(self, text)	从文本生成wordcloud。要删除重复项可设置collocations=False
generate_from_frequencies(self, frequencies)	根据单词和频率创建一个wordcloud
generate_from_text(self, text)	从文本生成wordcloud
process_text(self, text)	将长文本拆分为单词，消除停用词
recolor(self[, random_state, color_func, …])	重新着色。会比生成整个wordcloud快得多
to_array(self)	转换为numpy数组
to_file(self, filename)	导出图像文件

`ImageColorGenerator` 类

> **ImageColorGenerator**(image, default_color=None)

基于彩色图像的颜色生成器。 根据RGB图像生成颜色。单词将使用彩色图像中包围的矩形的平均颜色进行着色。 构造后，该对象充当可调用对象，可以作为color_func传递给词云构造函数或recolor方法。 image：类型nd-array，(height, width, 3) default_color：类型

tuple or None，默认为None

`wordcloud.random_color_func`

> **wordcloud.random_color_func**(word=None, font_size=None, position=None, orientation=None, font_path=None, random_state=None)

随机色调颜色生成。如果给定一个随机对象，则用于生成随机数。

示例

```
import wordcloud
# 实例化对象
test_English = wordcloud.WordCloud()
# 调用generate方法将文本生成wordcloud
# 还可调用generate_from_text方法同样实现
test_English.generate("I love China, my
motherland!")
# 图片可以事任何形状的
# 调用to_file方法导出图像文件，支
持.jpg、.png、.tif、.bmp等多格式
test_English.to_file("test_English.png")

# wordcloud默认不支持显示中文，中文会显示为方框。需先设置
好中文字体
# 以下字体为 电脑中的 的微软雅黑，拷贝至本项目文件中
font = r'msyh.ttf'
txt = "我爱中国，我的祖国！ "
```

```
test_Chinese =
wordcloud.WordCloud(font_path=font).generate(txt)
test_Chinese.to_file("test_Chinese.jpg")
```

5 Python 应用

5.1 lib 明细

1. 请求库：实现 HTTP 请求操作
    - `urllib` 是 `Python` 的内置库，一系列用于操作URL的功能
    - `re` 是 `Python` 的内置库，正则表达式，处理字符串
    - `requests` 是基于 `urllib` 编写的，库是用来在 `Python` 中发出标准的 `HTTP` 请求。它将请求背后的复杂性抽象成一个漂亮，简单的 `API` ，以便你可以专注于与服务交互和在应用程序中使用数据

> ```
> requests.get(url,params=None,**kwargs)
> ```
>
> **Response对象**
>
> 1. `URL`
> 2. `params`：字典或字符串格式作为参数增加到 `url` 中，是额外参数

3. `**kwargs`：代表十二个控制访问的参数
    - `data`：字典，字符串或文件对象，作为`Request`对象的内容
    - `json`： `JSON`格式的数据作为`Request`对象的内容
    - `headers`：字典，`HTTP`头部信息；目的是将请求伪装成诸如浏览器，使用`post`方法向服务器发起访问
    - `cookies`：字典或者`CookieJar`,`Request`中的`cookie`
    - `auth`：元组，支持`HTTP`认证功能
    - `files`：字典类型，向服务器传输文件
    - `timeout`： 设定超时时间，单位秒
    - `proxies`：字典类型，设定访问代理服务器，可以增加登录认证
    - `allow_redirects`： `True`/`False`,默认为`True`,重定向开关 //高级功能使用
    - `stream`：布尔值，默认为真，获取内容立即下载开关 //高级功能使用
    - `verify`：布尔值，默认为真，认证`SSL`证书开关 //高级功能使用
    - `cert`：保存本地`SSL`证书路径 //高级功能使用

```
response = requests.get(url,params=None,**kwargs)
```

Response对象的属性

    - `status_code`： HTTP请求返回的状态，200表示成功，404和其他表示失败；
    - `text`： 响应内容的字符串形式
    - `encoding`： 从响应内容的头部信息来推断编码形式
    - `apparent_encoding`： 从响应内容的内容信息来推断编码形式；
    - `content`： 将爬取到的响应内容的二进制形式还原成响应内容
    - `raise_for_status()`： HTTP的请求返回状态不是200则产生 `request.HTTPError`

2. 解析库：从网页中提取信息
    - `beautifulsoup4`： `html` 和 `XML` 的解析,从网页中提取信息，同时拥有强大的 `API` 和多样解析方式

      API地址 https://www.crummy.com/software/BeautifulSoup/bs4/doc/index.zh.html#find-all
    - `pyquery`： `jQuery` 的·Python·实现，能够以 `jQuery` 的语法来操作解析 `HTML` 文档，易用性和解析速度都很好(较 `bs4` 更加方便)
    - `lxml`： 支持 `HTML` 和 `XML` 的解析，支持 `XPath` 解析方式，而且解析效率非常高
    - `tesserocr`： 一个 `OCR` 库，在遇到验证码（图形验证码为主）的时候，可直接用 `OCR` 进行识别
3. 工具库
    - `json`： 既包含了将 `JSON` 字符串恢复成 `Python` 对象的函数，也提供了将 `Python` 对象转换成 `JSON` 字符串的函数

1. 使用 dumps 和 dump 函数进行编码，其中 dump 输出到文件fp中，其他功能与 dumps 一致

   常用参数为

   - Skipkeys： 默认值是 False，如果 dict 的 keys 内的数据不是 Python 的基本类型，设置为 False 时，就会报 TypeError 的错误。此时设置成 True，则会跳过这类 key
   - ensure_ascii： 默认值 True，如果 dict 内含有 non-ASCII 的字符，则会类似 \uXXXX 的显示数据，设置成 False 后，就能正常显示
   - indent： 应该是一个非负的整型，如果是0，或者为空，则一行显示数据，否则会换行且按照 inden t的数量显示前面的空白
   - separators： 分隔符，实际上是 (item_separator，dict_separator)的一个元组，默认的就是(',',':')；这表示 dictionary 内 keys 之间用“,”隔开，而 key 和 value 之间用“: ”隔开。
   - sort_keys： 将数据根据keys的值进行排序

2. 使用 loads 与 load 函数进行解码，与 dumps 相同， load 是从文件读入，其他和 loads 相同,常见参数可参照上文

- jieba：jieba 库是一种将句子分词的一个工具库

安装

- 自动安装 pip install jieba / pip3 install jieba ，推荐使用镜像安装：

```
 pip install -i
https://pypi.tuna.tsinghua.edu.cn/si
mple jieba
```

- 手动安装 github/jieba下载源码，将 jieba 目录放于当前目录或者 site-packages 目录
- 通过 import jieba 引用

api

```
cut(self, sentence, cut_all=False,
HMM=True) # 返回值是迭代器
# sentence：待分词字符串
# cut_all：全模式开启
# HMM：使用HMM，会多发现一些新词
lcut(self, sentence, cut_all=False,
HMM=True)  # 分词结果用列表返回
cut_for_search(self, sentence,
HMM=True) # 搜索引擎模式分词，会把
keyword都列出来
```

使用

```
seg_list = jieba.cut("我在看分词",
cut_all=False)
print(" ".join(seg_list))
```

5.2 天气查询

展示全国实时温度最低的十个城市气温排行榜的柱状图

天气API http://www.weather.com.cn/textFC/hb.shtml

5.3 招聘网站数据分析

生成职位名称的词云，https://search.51job.com/list/060000,000000,0000,00,9,99,Python,2,1.html

6 人工智能

6.1 OpenCV

`OpenCV` 是一个基于 `BSD` 许可（开源）发行的跨平台计算机视觉库，可以运行在 `Linux`、`Windows`、`Android` 和 `Mac OS` 操作系统上。它轻量级而且高效——由一系列 `C` 函数和少量 `C++` 类构成，同时提供了 `Python`、`Ruby`、`MATLAB` 等语言的接口，实现了图像处理和计算机视觉方面的很多通用算法(百度百科)。

```
pip install -i
https://pypi.tuna.tsinghua.edu.cn/simple opencv-
python
pip install -i
https://pypi.tuna.tsinghua.edu.cn/simple opencv-
contrib-python
```

API地址

常用操作：

1. 图片加载、显示和保存

- cv2.imread(filename, flags)： 读取加载图片

> `filepath`： 要读入图片的完整路径 `flags`： 读入图片的标志
>
> - `cv2.IMREAD_COLOR`： 默认参数，读入一副彩色图片，忽略 `alpha` 通道
> - `cv2.IMREAD_GRAYSCALE`： 读入灰度图片
> - `cv2.IMREAD_UNCHANGED`： 顾名思义，读入完整图片，包括 `alpha` 通道

- cv2.imshow(winname, mat)： 显示图片
- cv2.waitKey()： 等待图片的关闭
- cv2.imwrite(filename, img)： 保存图片

```
import cv2
# 读取图片,
img = cv2.imread(filename="cat.jpg",
flags=cv2.IMREAD_UNCHANGED)
# 显示图片，第一个参数为图片的标题
cv2.imshow(winname="image title", mat=img)
# 等待图片的关闭，不写这句图片会一闪而过
cv2.waitKey()
# 保存图片
# cv2.imwrite("Grey_img.jpg", img)
```

> cv2.waitKey()： 等待图片的关闭
>
> 可设置参数，为多少毫秒后自动关闭
>
> ```
> import cv2
>
> for file in filelist:
>     image = cv2.imread(file, flags=False)
>     cv2.imshow(winname="11",mat=image)
>  # 每隔1000ms显示一张图片
>  cv2.waitKey(1000)
> ```

2. 图像显示窗口创建与销毁

- cv2.namedWindow(winname, 属性)：创建一个窗口
- cv2.destroyWindow(winname)：销毁某个窗口
- cv2.destroyAllWindows()：销毁所有窗口

> winname作为窗口的唯一标识，如果想使用指定窗口显示目标图像，需要让cv2.imshow(winname)中的winname与窗口的winname需要保持一致。
>
> 窗口创建时可以添加的属性：
>
> - cv2.WINDOW_NORMAL：窗口大小可以改变（同cv2.WINDOW_GUI_NORMAL）
> - cv2.WINDOW_AUTOSIZE：窗口大小不能改变
> - cv2.WINDOW_FREERATIO：窗口大小自适应比例
> - cv2.WINDOW_KEEPRATIO：窗口大小保持比例
> - cv2.WINDOW_GUI_EXPANDED：显示色彩变成暗色
> - cv2.WINDOW_FULLSCREEN：全屏显示
> - cv2.WINDOW_OPENGL：支持OpenGL的窗口

```
img = cv2.imread("cat.jpg")
# 第二个参数为窗口属性
cv2.namedWindow(winname="title",
cv2.WINDOW_NORMAL)

# 如果图片显示想使用上面的窗口，必须保证winname一致
cv2.imshow(winname="title", img)
cv2.waitKey()
# 销毁
cv2.destroyWindow("title")
# 销毁所有窗口：cv2.destroyAllWindows()
```

3. 图片的常用属性的获取

- img.shape：打印图片的高、宽和通道数（当图片为灰度图像时，颜色通道数为1，不显示）
- img.size：打印图片的像素数目
- img.dtype：打印图片的格式

> 注意：这几个是图片的属性，并不是调用的函数，所以后面没有‘()’。

```
import cv2

img = cv2.imread("cat.jpg")
imgGrey = cv2.imread("cat.jpg", False)
print(img.shape)
print(imgGrey.shape)
#输出:
#(280, 300, 3)
#(280, 300)

print(img.size)
print(img.dtype)
#输出:
# 252000
# uint8
```

4. rectangle 标示

```
cv2.rectangle(img, (x1, y1), (x2, y2), (255,0,0), 2)
```

```
x1,y1 ------
|           |
|           |
|           |
--------x2,y2

cv2.rectangle(img, (x1, y1), (x2, y2), (255,0,0),
2)
img:  图像
(x1, y1):左上角点
(x2, y2):右下角点
(255,0,0):  RGB 颜色
2: thickness 参数表示矩形边框的厚度，如果为负值，如
CV_FILLED，则表示填充整个矩形
```

示例

导入图片

思路: 1.导入库 2.加载图片 3.创建窗口 4.显示图片 5.暂停窗口 6.关闭窗口

```
# 1.导入库
import cv2

# 2.加载图片
img = cv2.imread('a.png')
```

```
# 3.创建窗口
cv2.namedWindow('导入图片')

# 4.显示图片
cv2.imshow('导入图片',img)

# 5.暂停窗口
cv2.waitKey(0)

# 6.关闭窗口
cv2.destroyAllWindows()
```

在图片上添加人脸识别

思路: 1.导入库 2.加载图片 3.加载人脸模型 4.调整图片灰度 5.检查人脸 6.标记人脸 7.创建窗口 8.显示图片 9.暂停窗口 10.关闭窗口

```
# 1.导入库
import cv2

# 2.加载图片
img = cv2.imread('a.png')
# Haar特征分类器就是一个XML文件，该文件中会描述人体各个部位的Haar特征值。包括人脸、眼睛、嘴唇等等。
# 3.加载人脸模型,opencv官网下载
face =
cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
```

```
# 4.调整图片灰度:没必要识别颜色,灰度可以提高性能
gray = cv2.cvtColor(img,cv2.COLOR_RGB2GRAY)

# 5.检查人脸
# detectMultiScale检测出图片中所有的人脸，并将人脸用vector保存各个人脸的坐标、大小（用矩形表示）
# 参数 ：image表示的是要检测的输入图像
#        objects表示检测到的人脸目标序列
#         scaleFactor表示每次图像尺寸减小的比例
#        minNeighbors表示每一个目标至少要被检测到3次才算是真的目标(因为周围的像素和不同的窗口大小都可以检测到人脸),
#        minSize为目标的最小尺寸
#        maxSize为目标的最大尺寸
faces = face.detectMultiScale(gray)

# 6.标记人脸
for (x,y,w,h) in faces:
  # 里面有4个参数 1.写图片 2.坐标原点 3.识别大小 4.颜色 5.线宽
  cv2.rectangle(img,(x,y),(x+w,y+h),(0,255,0),4)

# 7.创建窗口
cv2.namedWindow('图片上人脸识别')

# 8.显示图片
cv2.imshow('图片上人脸识别', img)

# 9.暂停窗口
cv2.waitKey(0)
```

```
# 10.关闭窗口
cv2.destroyAllWindows()
```

调用摄像头

思路: 1.导入库 2.打开摄像头 3.获取摄像头实时画面 4.释放资源 5.关闭窗口

```
# 1.导入库
import cv2

# 2.打开摄像头
capture = cv2.VideoCapture(0)

# 3.获取摄像头实时画面
cv2.namedWindow('camera')
while True:
  #3.1 获取摄像头的帧画面
  ret,frame = capture.read()
  #3.2 显示图片(渲染画面)
  cv2.imshow('调用摄像头',frame)
  #3.3 暂停窗口
  # cv2.waitKey()中的数字代表等待按键输入之前的无效时间，单位为毫秒，在这个时间段内按键‘q’不会被记录，在这之后按键才会被记录
  if cv2.waitKey(5) & 0xFF == ord('q'):
    break
```

```
# 4.释放资源
capture.release()

# 5.关闭窗口
cv2.destroyAllWindows()
```

摄像头识别人脸

思路: 1.导入库 2.加载人脸模型 3.打开摄像头 4.创建窗口 5.获取摄像头实时画面 6.释放资源 7.关闭窗口

```
import cv2
import numpy as np
import os
import shutil



#采集自己的人脸数据
def generator(data):
    '''
    打开摄像头，读取帧，检测该帧图像中的人脸，并进行剪切、缩放
    生成图片满足以下格式:
    1.灰度图，后缀为 .png
    2.图像大小相同
    params:
        data:指定生成的人脸数据的保存路径
    '''
```

```
    name=input('my name:')
    #如果路径存在则删除路径
    path=os.path.join(data,name)
    if os.path.isdir(path):
        shutil.rmtree(path)
    #创建文件夹
    os.mkdir(path)

    # 1.加载人脸模型 创建一个级联分类器
    face =
cv2.CascadeClassifier('haarcascade_frontalface_def
ault.xml')

    # 2.打开摄像头
    capture = cv2.VideoCapture(0)

    # 3.创建窗口
    cv2.namedWindow('摄像头识别人脸')

    #计数
    count=1

    # 4.获取摄像头实时画面
    while capture.isOpened():
        # 4.1 获取摄像头的帧画面
        # 第一个参数ret 为True 或者False,代表有没有读取
到图片
        # 第二个参数frame表示截取到一帧的图片
```

```
        ret,frame = capture.read()  # 取出列表中的元
素

        if ret:
            # 5.2 图片灰度调整
            gray =
cv2.cvtColor(frame,cv2.COLOR_RGB2GRAY)
            # 5.3 检查人脸
            faces =
face.detectMultiScale(gray,1.3,5)
            print("发现{0}个目
标!".format(len(faces)))

            # 5.4 标记人脸
            for (x, y, w, h) in faces:
                # 里面有4个参数 1.写图片 2.坐标原点 3.
识别大小 4.颜色 5.线宽
                cv2.rectangle(frame, (x, y), (x +
w, y + h), (0, 0, 255), 4)
                #调整图像大小

 new_frame=cv2.resize(frame[y:y+h,x:x+w],(92,112))
                #保存人脸
                cv2.imwrite('%s/%s.png'%
(path,str(count)),new_frame)

                count+=1

             # 5.5 显示图片
```

```
            cv2.imshow('camera',frame)
            # cv2.waitkey(delaytime)-------
>returnvalue
            # 在delaytime时间内,按键盘, 返回所按键的
ASCII值;若未在delaytime时间内按任何键, 返回-1; 其
中,dalaytime: 单位ms;
            # note: 1. 当delaytime为0时,表示
forever,永不退回.
            # 2. 当按ecs键时,因为esc键ASCII值为27,所
有returnvalue的值为27,
            # ord('q'): 返回q对应的Unicode码对应的值,
q对应的Unicode数值为113
            # 0xFF是一个位掩码, 十六进制常数, 二进制值为
11111111, 它将左边的24位设置为0,把返回值限制在在0和255之
间。
            # ord(' ')返回按键对应的整数
            # 5.6 暂停窗口
            if cv2.waitKey(27) & 0xFF == ord('q'):
                break
    # 6.释放资源
    capture.release()
    # 7.关闭窗口
    cv2.destroyAllWindows()


#载入图像    读取ORL人脸数据库, 准备训练数据
def LoadImages(data):
    '''
    加载图片数据用于训练
```

```
    params:
        data:训练数据所在的目录，要求图片尺寸一样
    ret:
        images:[m,height,width]  m为样本数，height为
高，width为宽
        names：名字的集合
        labels：标签
    '''
    images=[]
    names=[]
    labels=[]

    label=0

    #遍历所有文件夹
    for subdir in os.listdir(data):
        subpath=os.path.join(data,subdir)
        #print('path',subpath)
        #判断文件夹是否存在
        if os.path.isdir(subpath):
            #在每一个文件夹中存放着一个人的许多照片
            names.append(subdir)
            print("subpath:"+subpath)
            print(names)
            #遍历文件夹中的图片文件
            for filename in os.listdir(subpath):

 imgpath=os.path.join(subpath,filename)
```

```
 img=cv2.imread(imgpath,cv2.IMREAD_COLOR)

 gray_img=cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
                #cv2.imshow('1',img)
                #cv2.waitKey(0)
                images.append(gray_img)
                labels.append(label)
            label+=1
    images=np.asarray(images)
    names=np.asarray(names)
    labels=np.asarray(labels)
    return images,labels,names



#检验训练结果
def FaceRec(data):

    #加载训练的数据
    X,y,names=LoadImages(data)
    #print('x',X)
    model=cv2.face.EigenFaceRecognizer_create()

    model.train(X,y)


    # 1.加载人脸模型 创建一个级联分类器
```

```
    face =
cv2.CascadeClassifier('haarcascade_frontalface_def
ault.xml')

    # 2.打开摄像头
    capture = cv2.VideoCapture(0)

    # 3.创建窗口
    cv2.namedWindow('摄像头识别人脸')

    # 4.获取摄像头实时画面
    while capture.isOpened():
        # 4.1 获取摄像头的帧画面
        # 第一个参数ret 为True 或者False,代表有没有读取
到图片
        # 第二个参数frame表示截取到一帧的图片
        ret,frame = capture.read()  # 取出列表中的元
素

        if ret:
             # 5.2 图片灰度调整
            gray =
cv2.cvtColor(frame,cv2.COLOR_RGB2GRAY)
            # 5.3 利用级联分类器鉴别人脸 检查人脸
            faces =
face.detectMultiScale(gray,1.3,5)

            # 5.4 标记人脸
            for (x, y, w, h) in faces:
```

```
                frame=cv2.rectangle(frame,(x,y),
(x+w,y+h),(255,0,0),2)  #蓝色

                roi_gray=gray_img[y:y+h,x:x+w]

                try:
                    #将图像转换为宽92 高112的图像
                    #resize（原图像，目标大小，（插值
方法）interpolation=，）
                    roi_gray=cv2.resize(roi_gray,
(92,112),interpolation=cv2.INTER_LINEAR)

                    params=model.predict(roi_gray)

 print('Label:%s,confidence:%.2f'%
(params[0],params[1]))
                    '''
                    putText:给照片添加文字
                    putText(输入图像，'所需添加的文
字'，左上角的坐标，字体，字体大小，颜色，字体粗细)
                    '''

 cv2.putText(frame,names[params[0]],(x,y-
20),cv2.FONT_HERSHEY_SIMPLEX,1,255,2)
                except:
                    continue

            # 5.5 显示图片
            cv2.imshow('camera',frame)
```

```
            # 5.6 暂停窗口
            if cv2.waitKey(27) & 0xFF == ord('q'):
                break
        # 6.释放资源
        capture.release()
        # 7.关闭窗口
        cv2.destroyAllWindows()


if __name__=='__main__':
    data='./face'
    # generator(data)
    FaceRec(data)
```

6.2 Face Recognition

本项目是世界上最简洁的人脸识别库，你可以使用Python和命令行工具提取、识别、操作人脸。

本项目的人脸识别是基于业内领先的C++开源库 dlib中的深度学习模型，用Labeled Faces in the Wild人脸数据集进行测试，有高达99.38%的准确率。但对小孩和亚洲人脸的识别准确率尚待提升。

首先安装 `dlib` https://pypi.org/simple/dlib/ (**dlib版本一定要与python版本匹配**)

```
pip install -i
https://pypi.tuna.tsinghua.edu.cn/simple
face_recognition
```

API地址： https://github.com/ageitgey/face_recognition/blob/master/README_Simplified_Chinese.md

6.3 pyttsx3

`pyttsx3`是一款非常简单的文本到语音的转换库，可以脱机工作，支持多种TTS引擎（`sapi5`、`nsss`、`espeak`），通过这个库可以非常方便的将文字转换成语音

```
pip install -i
https://pypi.tuna.tsinghua.edu.cn/simple pyttsx3
```

Engine的引擎API

方法签名	参数列表	返回值	简单释义
connect(topic : string, cb : callable)	topic：要描述的事件名称；cb:回调函数	→ dict	在给定的topic上添加回调通知
disconnect(token : dict)	token:回调失联的返回标记	Void	结束连接
endLoop()	None	→ None	简单来说就是结束事件循环
getProperty(name : string)	name有这些枚举值“rate, vioce,vioces,volumn	→ object	获取当前引擎实例的属性值

setProperty(name : string)	name有这些枚举值“rate, vioce,voices,volume	→ object	设置当前引擎实例的属性值
say(text : unicode, name : string)	text:要进行朗读的文本数据; name： 关联发音人，一般用不到	→ None	预设要朗读的文本数据，这也是“万事俱备，只欠东风”中的“万事俱备”
runAndWait()	None	→ None	这个方法就是“东风”了。当事件队列中事件全部清空的时候返回
startLoop([useDriverLoop : bool])	useDriverLoop:是否启用驱动循环	→ None	开启事件队列

```
import pyttsx3


def use_pyttsx3():
    # 创建对象
    engine = pyttsx3.init()
    # 获取当前语音速率
    rate = engine.getProperty('rate')
    print(f'语音速率: {rate}')
```

```
    # 设置新的语音速率 设置语速属性为当前语速减20
    engine.setProperty('rate', rate-20)
    # 获取当前语音音量
    volume = engine.getProperty('volume')
    print(f'语音音量: {volume}')
    # 设置新的语音音量, 音量最小为 0, 最大为 1
    engine.setProperty('volume', 1.0)
    # 获取当前语音声音的详细信息
    voices = engine.getProperty('voices')
    print(f'语音声音详细信息: {voices}')
    engine.setProperty('voice', 'zh')
    # 获取当前语音声音
    voice = engine.getProperty('voice')
    print(f'语音声音: {voice}')
    # 语音文本
    words = "你好, python"
#     with open(path, encoding='utf-8') as f_name:
#         words =
str(f_name.readlines()).replace(r'\n', '')
    # 将语音文本说出来
    engine.say(words)
    engine.runAndWait()
    engine.stop()


if __name__ == '__main__':
    use_pyttsx3()
```

6.4 PyAudio

API地址：http://people.csail.mit.edu/hubert/pyaudio/docs/

6.5 百度语音API

注册地址：https://ai.baidu.com/tech/speech

git示例：https://github.com/Baidu-AIP/speech-demo